DISSERTAÇÃO DE MESTRADO PROFISSIONAL

Detection of Obfuscated LOLBins Using Machine Learning and NLP Techniques

ÂNGELLO CÁSSIO VASCONCELOS OLIVEIRA

Orientador Prof. Dr. Daniel Chaves Café

Programa de Pós-Graduação Profissional em Engenharia Elétrica

DEPARTAMENTO DE ENGENHARIA ELÉTRICA
FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

DEPARTAMENTO DE ENGENHARIA ELÉTRICA

Detection of Obfuscated LOLBins Using Machine Learning and NLP Techniques

Detecção de LOLBins ofuscados usando técnicas de aprendizado de máquina e NLP

Ângello Cássio Vasconcelos Oliveira

Orientador: Prof. Dr. Daniel Chaves Café, unidade/UnB

PUBLICAÇÃO: PPEE.MP.083

BRASÍLIA-DF

UNIVERSIDADE DE BRASÍLIA
Faculdade de Tecnologia

DISSERTAÇÃO DE MESTRADO PROFISSIONAL

Detection of Obfuscated LOLBins Using Machine Learning and NLP Techniques

ÂNGELLO CÁSSIO VASCONCELOS OLIVEIRA

Orientador Prof. Dr. Daniel Chaves Café

Dissertação de Mestrado Profissional submetida ao Departamento de Engenharia Elétrica como requisito parcial para obtenção do grau de Mestre em Engenharia Elétrica

Banca Examinadora

Prof. Dr. Daniel Chaves Café, FT/UnB _____
Presidente

Prof. Dr. Georges Daniel Amvame Nze , FT/UnB _____
Examinador Interno

Dr. Rafael Salema Marques, University of Wolverhampton (UoW) _____
Examinador externo

FICHA CATALOGRÁFICA

OLIVEIRA, ÂNGELLO CÁSSIO VASCONCELOS
Detection of Obfuscated LOLBins Using Machine Learningand NLP Techniques [Distrito Federal] 2025.
xvi, 86 p., 210 x 297 mm (ENE/FT/UnB, Mestre, Engenharia Elétrica, 2025).
Dissertação de Mestrado Profissional - Universidade de Brasília, Faculdade de Tecnologia.
Departamento de Engenharia Elétrica

1. Living off the Land	2. Machine Learning
3. Obfuscated Commands	4. Cybersecurity
I. ENE/FT/UnB	II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

OLIVEIRA, A.C.V. (2025). *Detection of Obfuscated LOLBins Using Machine Learningand NLP Techniques*. Dissertação de Mestrado Profissional, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 86 p.

ÂNGELLO CÁSSIO VASCONCELOS OLIVEIRA
Depto. de Engenharia Elétrica (ENE) - FT
Universidade de Brasília (UnB)
Campus Darcy Ribeiro
CEP 70919-970 - Brasília - DF - Brasil

$$B > \frac{1}{n}\sum_{i=1}^{n} x_i$$

Be greater than the average.

Unknown

ACKNOWLEDGMENTS

I would like to begin my acknowledgments by expressing my deepest gratitude to my parents, whose unwavering support and education laid the foundation for reaching this pivotal stage in my career.

My sincere thanks go to my advisor, Daniel Café, for his steadfast guidance and support throughout this rigorous journey, particularly within the demanding timeframe of a master's degree.

I am also grateful to my friends Braulio and Reinaldo Gutierrez, whose encouragement to pursue admission at UnB and invaluable advice played a crucial role in my success.

A heartfelt thank you to Raphael Mota, Diogo Gonçalves, Rafael Salema, and Gianluigi Dal Toso; your support and camaraderie since 2018—when I first embarked on my journey in cybersecurity—made this challenging path far more manageable.

I am deeply appreciative of Saullo and Rafaela, whose steadfast friendship and encouragement inspired me to embrace new challenges and made me feel like part of a family.

Finally, I extend my eternal gratitude to my wife, Paloma Saldanha, who has stood by my side since high school, supporting every academic, professional, and personal endeavor along the way.

Without each and every one of you, none of this would have been possible.

RESUMO

O uso crescente de Living Off The Land Binaries (LOLBins) por grupos de Advanced Persistent Threat (APT) apresenta desafios significativos aos mecanismos de detecção tradicionais, pois essas técnicas exploram binários legítimos do sistema para executar atividades maliciosas. Este estudo avança o campo explorando a classificação de LOLBins, tanto em suas formas simples quanto ofuscadas, usando técnicas de aprendizado de máquina (ML) e processamento de linguagem natural (NLP). Com base em trabalhos anteriores, esta pesquisa incorpora métodos de normalização aprimorados e expande o conjunto de dados com comandos ofuscados, permitindo uma avaliação mais abrangente.

Por meio de experimentação sistemática, combinações de modelos de ML, incluindo Random Forest, Redes Neurais e Árvores de Decisão, foram testadas juntamente com métodos de NLP, como Bag-of-Words (BoW), TF-IDF e Doc2Vec. Algoritmos de balanceamento, incluindo Random Oversampling (ROS) e SMOTE, foram empregados para lidar com o desequilíbrio do conjunto de dados. Os resultados indicam que o Doc2Vec, emparelhado com técnicas de balanceamento robustas e modelos de ML otimizados, apresentou o melhor desempenho, alcançando alta precisão e pontuações de Coeficiente de Correlação de Matthews (MCC).

O estudo também se aprofunda nos desafios de detectar LOLBins ofuscados. Ao incorporar várias técnicas de ofuscação no conjunto de dados e empregar novas funções de normalização para desofuscar comandos, a pesquisa avalia o impacto das estratégias de pré-processamento na precisão da detecção. Embora a adição de dados ofuscados tenha expandido o conjunto de dados significativamente, também destacou as compensações entre a viabilidade computacional e o desempenho da classificação.

Este trabalho contribui para o campo da inteligência cibernética ao apresentar uma estrutura de detecção que aborda as táticas em evolução dos adversários. Ele ressalta a importância de combinar técnicas avançadas de PNL, pré-processamento eficaz e conjuntos de dados balanceados para melhorar as capacidades de detecção. As descobertas preparam o cenário para pesquisas futuras sobre métodos de normalização mais sofisticados e soluções escaláveis para lidar com o cenário dinâmico de ameaças à segurança cibernética.

Palavras-chave: LOLBins, Comandos Ofuscados, NLP, ML, Doc2Vec, TF-IDF, BoW, Segurança Cibernética, Inteligência Cibernética

ABSTRACT

The increasing use of Living Off The Land Binaries (LOLBins) by Advanced Persistent Threat (APT) groups presents significant challenges to traditional detection mechanisms, as these techniques exploit legitimate system binaries to execute malicious activities. This study advances the field by exploring the classification of LOLBins, both in their plain and obfuscated forms, using machine learning (ML) and natural language processing (NLP) techniques. Building upon previous work, this research incorporates enhanced normalization methods and expands the dataset with obfuscated commands, enabling a more comprehensive evaluation.

Through systematic experimentation, combinations of ML models, including Random Forest, Neural Networks, and Decision Trees, were tested alongside NLP methods such as Bag-of-Words (BoW), TF-IDF, and Doc2Vec. Balancing algorithms, including Random Oversampling (ROS) and SMOTE, were employed to address dataset imbalance, Results indicate that Doc2Vec, paired with robust balancing techniques and optimized ML models, delivered the best performance, achieving high accuracy and Matthews Correlation Coefficient (MCC) scores.

The study also delves into the challenges of detecting obfuscated LOLBins. By incorporating various obfuscation techniques into the dataset and employing new normalization functions to deobfuscate commands, the research evaluates the impact of preprocessing strategies on detection accuracy. While the addition of obfuscated data expanded the dataset significantly, it also highlighted trade-offs between computational feasibility and classification performance.

This work contributes to the field of cyber intelligence by presenting a detection framework that addresses the evolving tactics of adversaries. It underscores the importance of combining advanced NLP techniques, effective preprocessing, and balanced datasets to improve detection capabilities. The findings set the stage for future research into more sophisticated normalization methods and scalable solutions to tackle the dynamic landscape of cybersecurity threats.

Keywords: LOLBins, Obfuscated Commands, NLP, ML, Doc2Vec, TF-IDF, BoW, Cybersecurity, Cyber Intelligence

SUMÁRIO

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

In recent years, the use of Living Off The Land Binaries (LOLBins) has become a prevalent technique among Advanced Persistent Threat (APT) groups to evade detection (1). LOLBins (Living Off The Land Binaries and Scripts) (2) is a technique in which legitimate operating system binaries can be exploited by malicious individuals to subvert misconfigured systems (3). This technique, as elucidated by Ding et al. (4), is a furtive method used by attackers to harder their detection and is been increasingly seen as part of the APT's (Advanced Persistent Threats) Tactics, Techniques and Procedures (TTP) (5).

The use of legitimate tools stored within the system makes the detection of LOLBins attacks particularly challenging. Traditional security mechanisms, such as antivirus software and intrusion detection systems, typically rely on signature-based detection methods that can struggle to identify malicious actions carried out through benign, native binaries. Consequently, the detection of LOLBins requires active monitoring of system tools and command line activity, alongside advanced analytical techniques capable of discerning subtle deviations from normal behavior.

Barr-Smith et al. (6) conducted an investigation on 31,805,549 malware samples across multiple datasets, revealing an average prevalence of 9.41% for LOLBins techniques. When focusing on Advanced Persistent Threat (APT) usage, this figure increased significantly to 26.26%, with groups such as APT3, APT29, and APT33 relying heavily on these techniques.

In 2023, a relevant vulnerability was identified and exploited in Barracuda *Email Security Gateway* (ESG) (7) (8). This security failure consists in sending a malicious *.tar* file as an email attachment. When processed by the ESG system, the *.tar* file exploited a flaw in the validation and sanitization of file names within the system.

This security breach openned a vector to Remote Code Execution (RCE), which was abused using Living Off The Land Binaries (LOLBins) techniques. Analysing the Indicators Of Compromise (IOC) it was observed that the attackers used the code 1.1 to execute commands in the target through a *reverse shell*. With this purpose, the attackers used the *openssl* command to establish this reverse connection and *mkfifo* to create a named pipe to redirect the *sh* commands and its results between the target and the attacker's Command and Control (C2).

```
1 setsid sh -c "mkfifo /tmp/p;sh -i </tmp/p 2>&1| openssl s_client -quiet -connect
      107.148.149[.]156:8080 >/tmp/p 2>/dev/null;rm /tmp/p"
```

Code 1.1: Bash Script used in Barracuda exploit

In 2024, the government agency CISA (Cybersecurity and Infrastructure Security Agency), in coauthorship with other American agencies, published a report (9) that warns about the activities of the persistent group (APT) known as Volt Typhoon. An APT is an extremely organized group of cyber attackers, with the most diverse objectives, who choose to use low profile codes with a low noise rate to remain hidden, thus prolonging their presence on the target for as long as possible (10).

Volt Typhoon, specifically, focuses on entities related to its adversaries' critical infrastructures, such as the communications, energy, transportation and water sectors. In another report (11), from 2023, CISA shared some of the command lines used by the Volt Thyphoon group.

According to Code 1.2, these commands serve various purposes, such as redirecting traffic and enumerating the system, with the goal of finding credentials, all while utilizing native operating system binaries. For example, the group used the *netsh* command to redirect traffic from a specific port on all IP addresses (0.0.0.0) to another port on an internal IP address, using the TCP protocol. The *reg* command was used to query the Windows Registry for potential passwords and other valuable information.

```
# PortProxy
cmd.exe /c "netsh interface portproxy add v4tov4 listenaddress=0.0.0.0 listenport=9999 connectaddress=<rfc1918 internal ip address> connectport=8443 protocol=tcp"
cmd.exe /c "netsh interface portproxy add v4tov4 listenport=50100 listenaddress=0.0.0.0 connectport=1433 connectaddress=<rfc1918 internal ip address>"

# Registry enumeration
reg query hklm\software\OpenSSH
reg query hklm\software\OpenSSH\Agent
reg save hklm\sam ss.dat
reg save hklm\system sy.dat
```

Code 1.2: Exemplos de LOLBINS usados pelo Volt Thyphoon

Based on the widespread use of this technique by APT's, CISA also published a guide on the identification and mitigation of LOLBINS (12). In this guide, the agency references as a source of further information the GTFOBins (2) repository, for Linux, and its variables LOLBAS, LOLDRIVES and LOLBINS, for Windows, Windows Drivers and MacOS respectively. These repositories contain lists of commands native to the respective systems used for exploitation, most of which are used to escalate privileges on systems.

1.1 OBJECTIVES

The central goal of this study is to perform a binary classification of Obfuscated Linux commands, with the specific aim of identifying tactics associated with the use of LOLBins. To achieve this, combinations of several Machine Learning models will be employed, including Random Forest, Neural Networks and Decision Trees, concomitantly with the use of Natural Language Processing (NLP) techniques, such as Bag of Words, TF-IDF and Doc2Vec. Furthermore, to overcome the challenge of data imbalance, artificial database balancing algorithms will be implemented.

For this purpose, Thuy Ngan's master's thesis (13) was used as a basis, which carried out the binary classification and risk levels of Linux commands. To achieve his objective, Ngan set up a database with Linux commands extracted from GitHub, which he considered benign, and commands obtained through two honeypots, which he classified as malignant.

Thus, the analysis focused on commands originating from the Linux operating system, with a significant portion of the data originating from records in the file "*.bash_history*", the same way portion used by Ngan (13) in his work, and the list of commands from the GTFOBins website (2). This restricted the information available to command text only. Given this contextual limitation, the solution proposed by Ngan (13) was used, which involved analyzing a window of N previous commands to provide more context to the lack of detailed data about each command.

As highlighted by Kotsiantis et al. (14), the challenge of unbalanced data is increasingly common, especially in practical situations that involve the detection of rare but relevant events. This scenario is similar to the context of identifying malicious commands or malwares. Due to the limited and relatively small list of LOLBins, it was necessary to employ artificial dataset balancing techniques given that these commands represented only 1,601% of the total. To mitigate this disparity, several techniques were explored, including *Random Undersampling*, *Random Oversampling* and *Near Miss*, aiming to balance data distribution and improve the effectiveness of detection models.

However, for a better analysis of LOLBins, it was necessary to consider not only Machine Learning models and Artificial Balancing algorithms, but also the Natural Language Processing (NLP) algorithms used to represent the command's strings. This interaction between different approaches directly influenced the final metrics of each test.

Then the research aimed to enhance the detection capabilities by addressing the challenge of command obfuscation. Obfuscation techniques are commonly employed by attackers to disguise their activities, making it harder for security tools to recognize malicious commands. With this in mind, the model that demonstrated the best performance in earlier tests was fine-tuned using the Cross Validation technique.

Following this fine-tuning process, the LOLBins commands were subjected to various obfuscation techniques, automating the creation of a new database, before being submitted to the model. Firstly, the commands were pre-processed using the methods developed in the previous study and once again with a minor change in only one function. Subsequently, a new pre-processing function was introduced to specifically identify and decode the obfuscated commands.

1.2 CONTRIBUTIONS OF THIS WORK

The primary contributions of this work are summarized as follows:

- **Increasing of an Existing Dataset:** As a base point for the beginning of this work, the benign portion of Ngan's (13) database was used in conjunction with the database from the GTFOBins (2) repository, meaning that the old dataset was then increased with new portions. Then research expanded the existing dataset by introducing obfuscation techniques into the LOLBins commands.
- **Enhanced Machine Learning Models:** By comparing and optimizing various machine learning models—such as Random Forest, Decision Trees, and Neural Networks—across different Natural Language Processing (NLP) techniques, this work identifies the most effective approaches for detecting LOLBins. For this purpose it was implemented, as well, various artificial data balancing

techniques, such as Random Oversampling, SMOTE, and Near Miss, to address the issue of imbalanced datasets. Finally, the use of cross-validation and fine-tuning ensured that the models are robust and well-suited for practical application.

- **Comprehensive Detection Framework:** Significant improvements were made to the preprocessing of command-line data, proposed by Ngan(13), particularly in handling file paths, environmental variables, and other text representations. Additionally, specialized functions were developed to decode obfuscated commands, thereby enhancing the detection capabilities of the models.

1.3 STRUCTURE OF THE THESIS

After the Chapter 1 - **Introduction**, this thesis is organized into four main chapters. Firstly, In Chapter 2 - **Concepts and Related Work**, it'll be discussed the theoretical framework that underpins the research, including a review of relevant literature for contextualizing the problem and covering the concepts around the research.

Then, In Chapter 3 - **Methodology**, the methods and techniques used to conduct the study will be detailed, covering everything from data collection to the analytical procedures employed.

As for the Chapter 4 - **Data Analysis and Results**, on it will be presented and discussed the results obtained, interpreting the findings in light of the established hypotheses and objectives. Finally, in Chapter 5 - **Conclusion**, a summary of the main conclusions of the study and suggested directions for future work will be presented.

2 CONCEPTS AND RELATED WORK

In this chapter, we explore the key concepts and existing literature that form the foundation of this research. Understanding the current state of art of of the field is crucial for contextualizing the contributions of this work and highlighting the gaps that our approach seeks to address. The chapter is structured to first explain the theoretical concepts guiding this research, followed by a review of the related work in the area of malware and code detection, mainly using machine learning.

2.1 THEORETICAL CONCEPTS

This section introduces the key theoretical concepts that underpin this research, including machine learning (ML) models, natural language processing (NLP), artificial data balancing techniques, evaluation metrics, and model fine-tuning. By anchoring the study in these concepts, we establish a solid foundation for understanding the role of machine learning in detecting malicious code.

We focus on three critical pillars for effective ML training: selecting the appropriate ML model, applying NLP techniques, and ensuring proper dataset balancing. The selection of the right model depends on identifying the most relevant metrics for the task. Once these pillars are established, the model must be fine-tuned to optimize its performance and achieve the best possible results.

2.1.1 Machine Learning Algorithms

As first explained by Arthur Samuel in 1959 (15), Machine Learning (ML) is a field that enables computers to learn without being explicitly programmed. Broadly speaking, ML algorithms can be categorized into two primary types: supervised and unsupervised learning, as outlined by Kong et al. (16).

> Usually, we classify machine learning into two main categories, i.e. supervised learning and unsupervised learning. The supervised learning as the name suggested, is that during the training of the algorithm, we know the correct label, which means we know the answers of the problem, this prior information will be used in the training. Within the supervised learning, depending on the nature of the output, we can divide the algorithms into classification and regression. For example, if we are asked to design an algorithm to recognize apple and oranges, and we know which object is apple or orange, then this problem is the classification problem, since the output will be categorical data, either orange or apple. The different objects are usually called classes. [...]. The other category of the algorithms are unsupervised learning, which means we don't have the luxury of the labels. [...]. Then this will be a clustering problem, which you need to use some of the hidden characteristics of the objects to group them together. Dimensionality reduction is a group of algorithms within the category of unsupervised learning to reduce the higher dimension problems into lower dimension ones. (16)

This distinction is based on whether the data is labeled or unlabeled, as well as the intended use of the final model. In this work, we focus on Supervised Learning, as our dataset is labeled, with the primary objective being to classify the inputs. Specifically, our dataset consists of legitimate commands and malicious LOLBins commands, with the main goal being to accurately differentiate between the two.

Classification can be further subdivided into binary and multi-class classification (17). In our approach,

we chose binary classification, where commands are categorized as either malicious or benign. However, in Ngan's work (13), a multi-class classification was employed, as the objective was to categorize malicious commands in a honeypot according to different risk levels, which will be discussed in more detail later.

To achieve optimal results, we applied several machine learning algorithms: k-Nearest Neighbors (KNN), Decision Tree (DT), Random Forest (RF), Linear Regression (LR), Support Vector Machine (SVC) and Neural Networks (NN). Each algorithm processes data differently due to its unique implementation, which can lead to variations in performance, resulting in better or worse outcomes depending on the data characteristics.

2.1.1.1 k-Nearest Neighbors

Conway (18) defined KNN as the simplest algorithm that was covered in his book.

> It's arguably the most intuitive of all the machine learning algorithms that we present in this book. Indeed, the simplest form of k-nearest neighbors is the sort of algorithm most people would spontaneously invent if asked to make recommendations using similarity data: they'd recommend the song that's closest to the songs a user already likes, but not yet in that list. That intuition is essentially a 1-nearest neighbor algorithm. The full k-nearest neighbor algorithm amounts to a generalization of this intuition where you draw on more than one data point before making a recommendation. (18)

The KNN algorithm plots all elements on a graph based on their features. For each new, unknown item, it calculates the distances to the nearest points, identifying the closest items to determine the classification. K is the number of the closest neighbors chosen for this measurement.

As explained in the book *Practical Statistics for Data Scientists* (19), the distance—also referred to as similarity or proximity—between records is mathematically calculated. One of the most common methods for measuring the distance between two vectors is the Euclidean distance. The Euclidean distance between two points $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_n)$ in an n-dimensional space is given by:

$$d_{\text{Euclidean}}(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

While the Euclidean distance represents the straight-line distance between two points, the Manhattan distance, also known as the L1 distance or taxicab, calculates the distance by summing the absolute differences of their coordinates. Unlike Euclidean distance, which measures the shortest path, Manhattan measures the distance between two points based on total horizontal and vertical displacements (20):

$$d_{\text{Manhattan}}(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

The Minkowski distance is another method for calculating distances and is commonly used in facial measurements for Computer Vision applications (21). This distance metric generalizes both the Euclidean and Manhattan distances by introducing a parameter p, allowing it to adapt to various measurement needs. When $p = 2$, it corresponds to the Euclidean distance; when $p = 1$, it becomes the Manhattan (or taxicab)

distance; and for $p = \infty$, it is equivalent to the Chebyshev distance (22):

$$d_{\text{Minkowski}}(x, y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}}$$

The final distance metric used by KNN is the Weighted KNN. In this approach, the distance between points is weighted, typically inversely proportional to their distance. As explained by Bicego and Loog (23), closer neighbors have a greater influence on the classification result. The weight for each neighbor is calculated using the formula $w_i = \frac{1}{d(x,y)}$, where $d(x, y)$ can be any of the previously mentioned distance metrics, with the choice of distance metric directly impacting the final result. For example, using Euclidean distance with weights, where k is the number of nearest neighbors:

$$d_{\text{Weighted}}(x, y) = \sum_{i=1}^{k} \frac{1}{d_{\text{Euclidean}}(x, y_i)}$$

Since these distances are mathematically calculated, it's essential to ensure that all features are in numerical form. There are two primary methods for this: One-Hot Encoding and Label Encoding. In One-Hot Encoding, each categorical variable in a feature is represented as a new column with binary values (24). Label Encoding, on the other hand, assigns a unique numerical value to each category (25).

Not only must the features be numerical, but they also need to be on a comparable scale. For example, consider a dataset with features such as a person's height, birth year, and salary. These values are measured on different scales—e.g., a height of 1.70 meters, birth year 1991, and salary of 15,000.00. If these features are not scaled proportionally, the feature with the largest range (in this case, salary) will likely dominate and have a greater influence on the model's results.

To scale features, one can apply either standardization or normalization techniques (26). While both are often referred to as normalization, Z-score normalization is specifically known as standardization. Standardization transforms data to have a mean of 0 and a standard deviation of 1, following the Z-score formula:

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

where: - x is the data point, - μ is the mean of the dataset, - σ is the standard deviation of the dataset.

There are several ways to normalize a feature. For instance, Akilli and Atil (27) studied eight normalization techniques, including *Z-Score, Min-Max, D-Min-Max, Median, Sigmoid, Decimal Scaling, Median and MAD, and Tanh-Estimators*. The most commonly used method is Min-Max Normalization, which scales data to a specified range, typically between 0 and 1. For a data point x, with x_{min} and x_{max} as the minimum and maximum values in the dataset, Min-Max Normalization is defined as:

$$x_{\text{scaled}} = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}}$$

Other normalization techniques include Mean Normalization, Decimal Scaling Normalization, and Max Absolute Scaling (28). Mean Normalization centers data around 0 by subtracting the mean μ and scaling by the range:

$$x_{\text{normalized}} = \frac{x - \mu}{x_{\text{max}} - x_{\text{min}}}$$

Decimal Scaling Normalization, in turn, adjusts data by shifting the decimal point so that the largest absolute value in the dataset is less than 1. This is achieved by scaling values by a power of 10, where j is the smallest integer such that $|x_{\text{scaled}}| < 1$ for all values:

$$x_{\text{scaled}} = \frac{x}{10^j}$$

This normalization technique is particularly useful when the relative magnitude of values is more significant than their exact scale. For example, Sinsomboonthong found that *"the normalization efficiency of decimal scaling for k-nearest neighbor was higher than that of Z-score, median, and min-max for k-nearest neighbor, Naïve Bayes, and artificial neural networks using classification accuracy on six datasets"* (29). Thus, the choice of normalization technique can directly impact the model's performance.

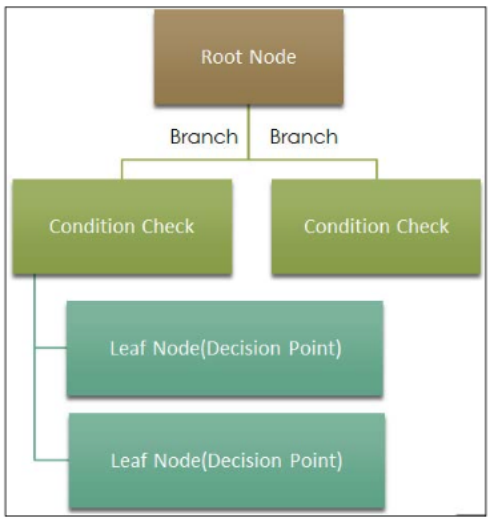
Finally, selecting an appropriate k value is crucial to optimizing KNN model performance. Research, such as that by Romero-del-Castillo et al. (30), has focused on finding the optimal k value, demonstrating its importance in achieving accurate results.

2.1.1.2 Decision Tree & Random Forest

At a high level, a Decision Tree (DT) is a conjunction of rules arranged in a hierarchical order (31). In other words, it can be thought of as a series of if-then-else rules, which makes the algorithm, as Li et al. (32) describe, *"easy to implement, highly interpretable, fully compatible with human intuitive thinking, and capable of handling large-scale data."*

To gain a better understanding of Decision Trees (DT), it's essential to become familiar with some key terminologies. First, there is the *root node*, which serves as the starting point of the tree. From the root node, *branches* extend to represent condition checks, following a series of if-else statements as previously mentioned. These branches may lead to additional condition checks or to *leaf nodes*, which are the final decision points where classification or regression outcomes are determined. This structure is illustrated by Gollapudi in Figure 2.1.

Figure 2.1: Diagram of a decision tree.



Source: Adapted from Gollapudi (31).

The algorithm uses specific metrics to determine the path taken at each decision point. As explained by Kirk (33), the three most commonly used metrics for this purpose are Information Gain, Gini Impurity, and Variance Reduction:

- **Information Gain** (IG) measures the reduction in entropy, which quantifies the impurity or disorder in a dataset, after the dataset is split based on a particular attribute. It is defined as:

$$IG(S, A) = H(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} H(S_v)$$

  where $H(S) = -\sum_{i=1}^{n} p_i \log_2(p_i)$ represents the entropy of the original dataset S (34). In this formula, S_v denotes the subset of S for which the attribute A has a specific value v, and $|S_v|$ and $|S|$ are the sizes of the subsets S_v and S, respectively.

- **Gini Impurity** (G), which differs from GINI coefficient (33), measures the probability of incorrectly classifying a randomly chosen element from the dataset if it were randomly labeled according to the distribution of labels in the dataset (35). The impurity function i_{Gini} is calculated as:

$$i_{\text{Gini}}(\mathbf{u}) = \sum_{i=1}^{d} \frac{u_i}{\|\mathbf{u}\|_1} \left(1 - \frac{u_i}{\|\mathbf{u}\|_1}\right)$$

  For a given vector $\mathbf{u} = (u_1, u_2, \ldots, u_d)$, $\|\mathbf{u}\|_1$ is the L1 norm of vector $\mathbf{u}$, which is the sum of its components.

- **Variance Reduction** (VR) is mainly applied in continuous decision trees, where the objective is to reduce the dispersion within data splits to produce more predictable, consistent outputs (33). Variance reduction does not apply to classification tasks but is relevant for continuous outputs. For a given split on attribute A, VR is calculated as:

$$\xi = \mathbb{E}(X_{1j}) - \mathbb{E}(X_{2j}) = \mu_1 - \mu_2$$

where $\mathbb{E}(X_{1j})$ and $\mathbb{E}(X_{2j})$ are the expected values (means) for subsets formed by the split.

As explained by Garreta and Moncecchi (36), one of the limitations of decision trees (DTs) is that, once trained, they cannot generate new condition checks dynamically. To address this limitation, Random Forest (RF) was introduced by Breiman (37). This algorithm uses ensemble methods to combine multiple decision trees into a "forest,"thereby improving the model's overall robustness and reducing the likelihood of overfitting.

2.1.1.3 Logistic Regression

According to Isoni (38), Linear Regression is one of the simplest algorithms within the Generalized Linear Models (GLM) framework.

> Linear regression is the simplest algorithm and is based on the model:
>
> $$h_\theta\left(x^{(i)}\right) = \theta_0 + \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)} + \cdots = \sum_{j=0}^{M-1} \theta_j x_j^{(i)}, \quad \forall i \in \{0, \ldots, N-1\}$$
>
> The cost function and update rule are:
>
> $$J = \frac{1}{2} \sum_{i=0}^{N-1} \left(y_i - h_\theta\left(x^{(i)}\right)\right)^2 \quad \rightarrow \quad \frac{\partial J}{\partial \theta_j} = \left(y_i - h_\theta\left(x^{(i)}\right)\right) x_j^{(i)}, \quad \forall j \in \{0, \ldots, M-1\}$$
>
> (38)

GLMs attempt to establish a linear relationship between the target variable and input features, which can be expressed as:

$$y_i = \sum_{j=0}^{M-1} \theta_j x_j^{(i)} + \epsilon_i = h_\theta\left(x^{(i)}\right) + \epsilon_i \quad \forall i \in \{0, \ldots, N-1\}$$

In this formula, y_i represents the target variable, θ_j are the model parameters, $x_j^{(i)}$ denote the feature values for the i-th instance, and ϵ_i is the error term (38). Linear Regression is suitable for regression tasks where the goal is to predict continuous values. However, for classification tasks, particularly binary classification, Logistic Regression (LR) is used (39).

As described by Schober and Vetter (40), Logistic Regression builds upon Linear Regression by establishing a relationship between an independent variable X and the probability of a binary outcome. This is accomplished by constraining predicted probabilities within the range of 0 to 1, using a linear relationship with the logit (log-odds) of the outcome. The logistic regression model is defined as:

$$\ln\left(\frac{p}{1-p}\right) = b_0 + b_1 X$$

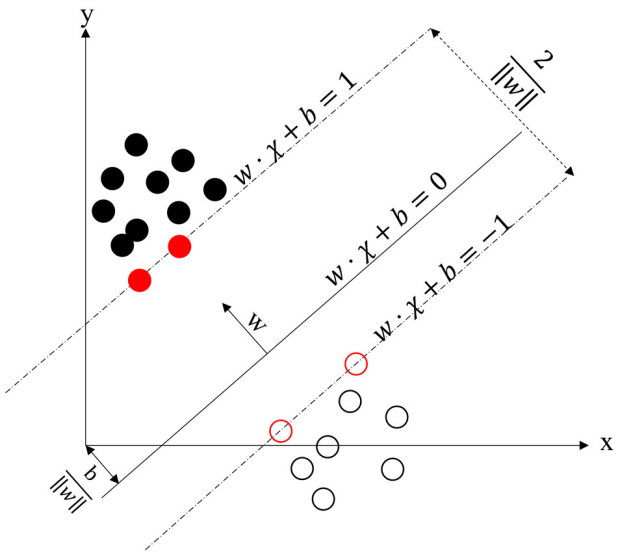
where b_0 is the intercept and b_1 represents the slope. Solving this equation for probability p results in a sigmoidal relationship with X, ensuring that the probabilities are limited between 0 and 1.

2.1.1.4 Support Vector Machine

In Support Vector Machines (SVM), each feature is represented as a coordinate in an n-dimensional space, where n is the number of features. The algorithm's objective is to find the optimal hyperplane (or line, in the case of two dimensions) that best separates the classes by maximizing the margin between data points of different classes, thereby establishing a clear decision boundary (41).

SVM can use either linear boundaries or apply various kernel functions to capture more complex relationships between features. When the dataset is linearly separable, multiple hyperplanes may exist, but SVM identifies the one that maximizes the geometric margin between classes, ensuring the greatest separation (42), as shown in Figure 2.2.

Figure 2.2: Support Vector Machine principles.



Source: Adapted from Guo and Li's article (42).

While SVM is fundamentally a binary classifier, it can handle multiclass classification problems by employing strategies like One-vs-All and One-vs-One (43). For each binary classification, the goal is to find the optimal hyperplane that best separates the two classes, labeled $+1$ and -1. As discussed in *"Large Scale Machine Learning with Python"* (44), this is achieved by minimizing a cost function that balances two components: a regularization term, which controls model complexity, and a loss term, which handles classification errors.

- **Regularization Term**: $\frac{\lambda}{2}\|w\|^2$, where w is the weight vector defining the hyperplane, and λ is a non-negative regularization parameter. This term penalizes large values of w, promoting a simpler model that avoids overfitting by controlling the width of the margin.

- **Loss Term (Slack Variable)**:

$$\frac{1}{n}\sum_{i=1}^{n}\max(0, 1 - y_i(w \cdot X_i + b))$$

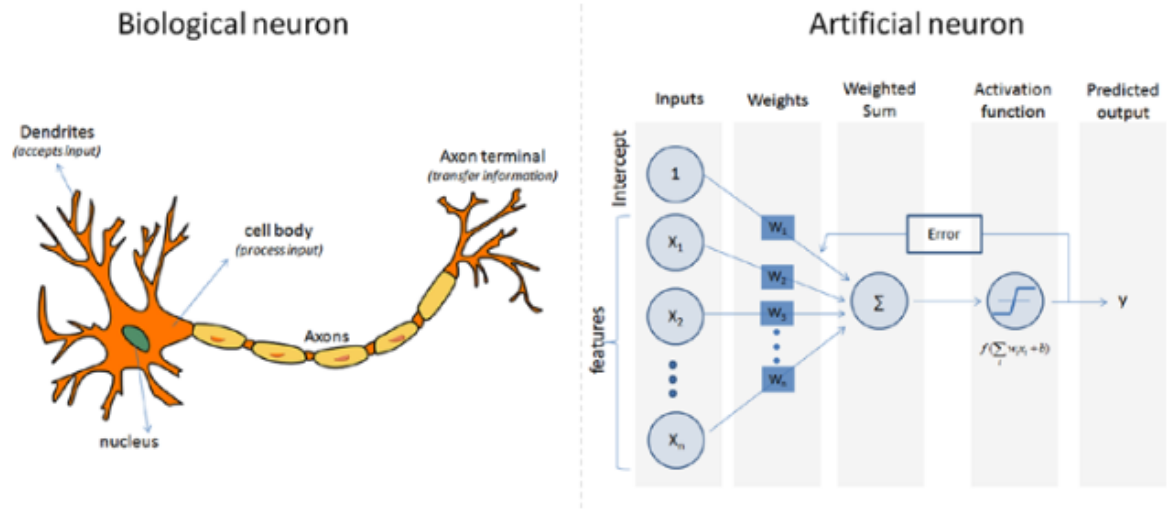
  where y_i represents the true class label ($+1$ or -1) for each data point X_i, and b is the bias or offset. This term measures the classification error, applying a margin constraint of $1 - y_i(w \cdot X_i + b)$. If a point is on the correct side of the margin, this term contributes 0 to the cost. Otherwise, it contributes an amount proportional to the distance of the point from the margin boundary.

By minimizing this cost function, SVM achieves a balance between maximizing the margin (controlled by $\|w\|$) and minimizing misclassification errors (managed through the loss term).

2.1.1.5 Neural Network

Neural Networks (NN) were developed to simulate the functioning of biological neurons. The concept of an artificial neuron, known as the perceptron, was first introduced in 1943 by McCulloch and Pitts (45) and was implemented by Frank Rosenblatt in 1958.

Figure 2.3: Biological vs. Artificial Neuron.



Source: Adapted from Swamynathan's book (45).

Figure 2.3 illustrates a comparison between a human neuron and a perceptron. In a perceptron, each input x_i is assigned a weight w_i, with an additional intercept term, or bias, w_0. The perceptron computes a weighted sum of the inputs and passes it through an activation function f to produce the output y:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + w_0\right)$$

where y is the output, f is the activation function, w_i represents the weight for each input x_i, and w_0 is the bias term. The activation function f introduces non-linearity into the model, allowing it to solve more complex problems than simple linear models. The output y is typically binary, with $y = 1$ if the weighted sum exceeds a specified threshold and $y = 0$ otherwise. Although multiple activation functions exist, the four most commonly used are Sigmoid, Hyperbolic Tangent, Hard Limiting Threshold, and Linear. Table 2.1, as presented in "Neural Network Programming with Java"(46), shows these four functions along with their corresponding equations and charts.

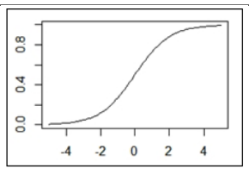
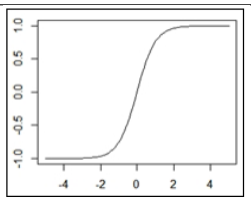
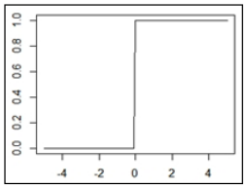
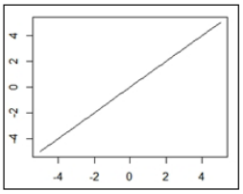
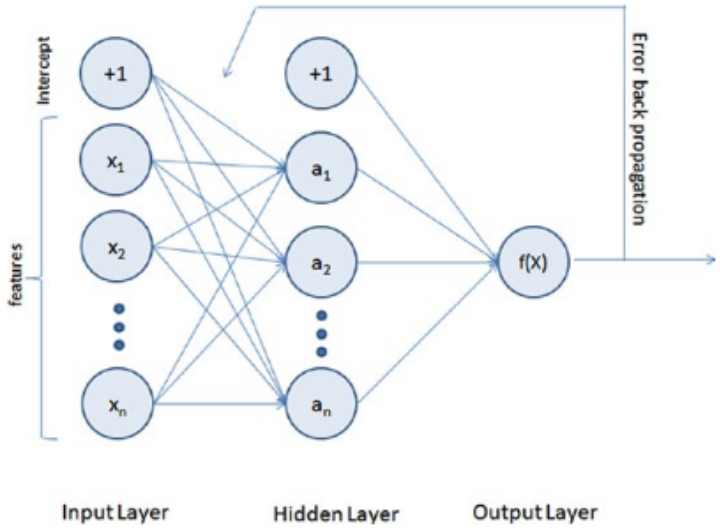
Function	**Equation**	**Chart**
Sigmoid	$f(x) = \frac{1}{1+e^{-x}}$	
Hyperbolic tangent	$f(x) = \frac{1-e^{-x}}{1+e^{-x}}$	
Hard limiting threshold	$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 1 \end{cases}$	
Linear	$f(x) = x$	

Table 2.1: Activation Functions and Their Graphs

However, a single perceptron has limitations as it only works well for linearly separable classes. To address this limitation, multilayer perceptrons (MLPs) were developed, allowing the network to handle more complex, non-linear classifications by including multiple hidden layers (47), as seen in Figure 2.4.

Figure 2.4: Multilayer Perceptron Representation.



Source: Adapted from Swamynathan's book (45).

As explained by Swamynathan (45), the formula for an MLP with a single hidden layer is given by:

$$y = f\left(\sum_{j=0}^{M} w_j^{(2)}\, g\left(\sum_{i=0}^{n} w_{ij}^{(1)} x_i + w_0^{(1)}\right) + w_0^{(2)}\right)$$

In this formula, $w_{ij}^{(1)}$ represents the weight of the connection from input x_i to the j-th node in the hidden layer, while $w_j^{(2)}$ is the weight of the connection from the j-th hidden node to the output layer. There are two bias terms here: $w_0^{(1)}$ for the hidden layer and $w_0^{(2)}$ for the output layer. M and n denote the number of hidden nodes and input features, respectively.

According to Bilski et al. (48), each neural network can be trained to perform a specific task using two main processes: feedforward and backpropagation. In a feedforward neural network (FNN), data flows in one direction, from the input layer through the hidden layers to the output layer. Backpropagation, on the other hand, is a process in which the model calculates the error after each training iteration and adjusts the weights starting from the output layer back to the input layer (49).

As outlined by Soares and Souza (46), backpropagation utilizes gradient descent to update weights in a neural network with hidden layers. For a given weight w_{ij}, the adjustment is determined by:

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} = -\eta \delta_i x_j$$

where E is the error, η is the learning rate, and x_j is the input to the neuron.

The backpropagation error δ_i for a neuron i differs depending on whether it is in the output layer or a hidden layer. For an output layer neuron, δ_i is calculated as:

$$\delta_i = (o_i - t_i) f'(h_i)$$

having o_i as the actual output, t_i as the target output, $f'(h_i)$ as the derivative of the activation function, and h_i as the weighted sum of inputs. On the other hand, for a hidden layer neuron, δ_i is computed as:

$$\delta_i = \sum_l \delta_l w_{li} f'(h_i)$$

where l indexes neurons in the next layer, and w_{li} is the weight connecting neuron i in the hidden layer to neuron l in the following layer. This adjustment enables the neural network to learn by minimizing the overall error across layers.

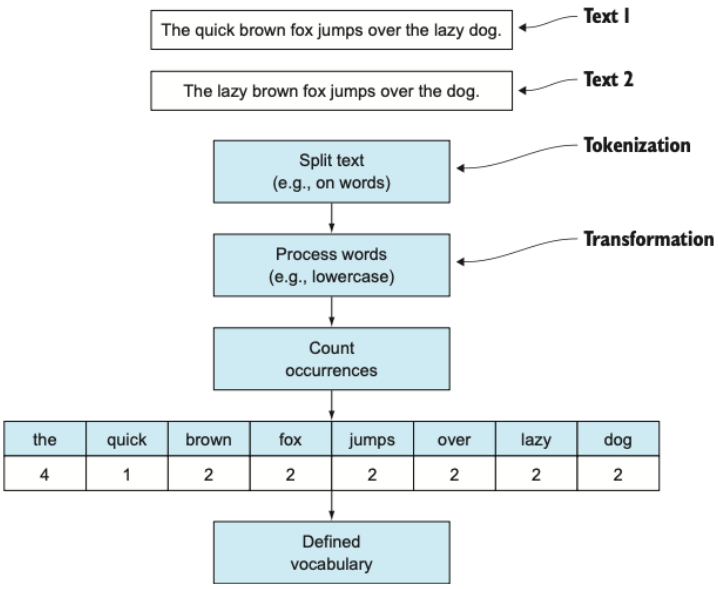
2.1.2 Natural Language Processing

As discussed in the KNN section, computers do not inherently understand words, instead they process information in numerical form. To address this limitation, Natural Language Processing (NLP) algorithms were developed (50). NLP allows ML algorithms to recognize and analyze patterns in text, enabling them to interpret and process human language (51). In this work, for example, NLP was used to recognize command-line inputs as an example of structured text. Various NLP algorithms are available, and in this work, three were utilized: Bag of Words (BOW) , Term Frequency-Inverse Document Frequency (TF-IDF), and Doc2Vec.

> In the BOW model the documents are represented by vectors in which each dimension corresponds to a word or group of words, generating vectors with a very high dimensionality. It is an exclusively lexical representation that relies on metrics based on frequency to determine the values associated with each dimension of the vector. (52)

In other words, BOW is one of the simplest NLP algorithms. It represents text as a set, or "bag,"of tokens, discarding any information about word order, as all words are treated as separate, unordered items. As explained by Brink et al. in their book, the best way to understand BOW is by dividing it into two stages: tokenization and vectorization (39).

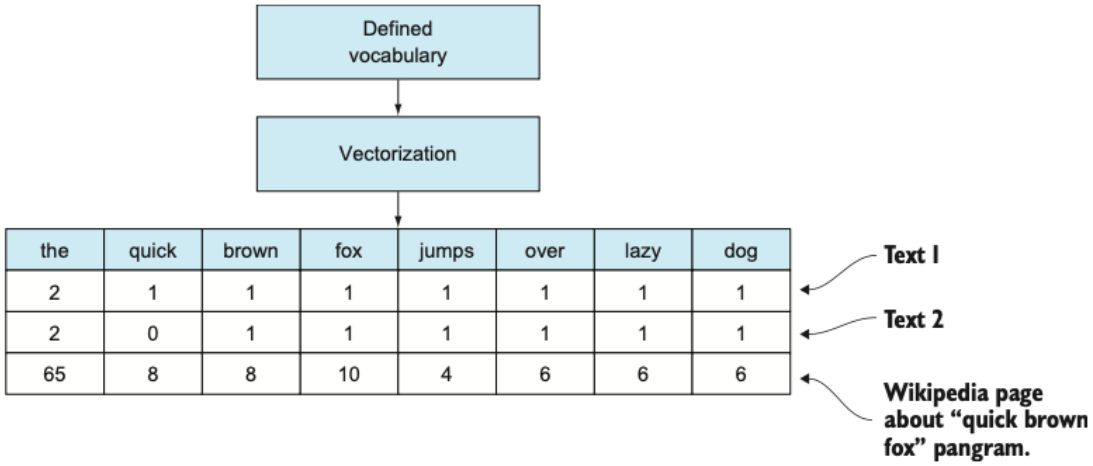
Figure 2.5: The initial steps in the bag-of-words extraction algorithm.



Source: Adapted from Brink's book (39).

As shown in Figure 2.5, the process of tokenization involves dividing the text into individual units, or list of terms, known as tokens (53). In the Figure 2.5, the text is split by spaces. Using single words as tokens is referred to as *unigrams*, though in some cases, performance can improve by using *bigrams*, *trigrams*, or *n-grams*. As detailed in the Methodology section, this work tests both *unigram* and *trigram* representations to capture patterns in command lines.

Figure 2.6: The vectorization process in BOW implementation.



Source: Adapted from Brink's book (39).

After tokenization, the next step is vectorization, where features are generated based on the dictionary of tokens. As elucidated by Khomsah et al. (54), a significant challenge with the BOW approach arises

when applied to a large corpus, as it can result in an enormous and sparse matrix that becomes computationally intensive to manage.

As mentioned earlier, with BOW, all tokens are combined into a "bag,"losing any information about their order or structure. To address this limitation, one commonly used algorithm is TF-IDF. This algorithm assigns weights to measure the importance of each token within a document by calculating the product of the term frequency (TF) and the inverse document frequency (IDF) (39).

> The main idea of TF-IDF is that, if a word appears frequently in a document, but less frequently in other documents, the word has a greater effect for distinguishing the document and expressing the core content of the document, and therefore has a higher weight (55)

The term frequency (TF) reflects how often a token appears within a single document, while the inverse document frequency (IDF) considers the entire corpus, assigning lower weights to commonly occurring words. Mathematically, given a corpus with D documents, TF-IDF can be defined as follows (56):

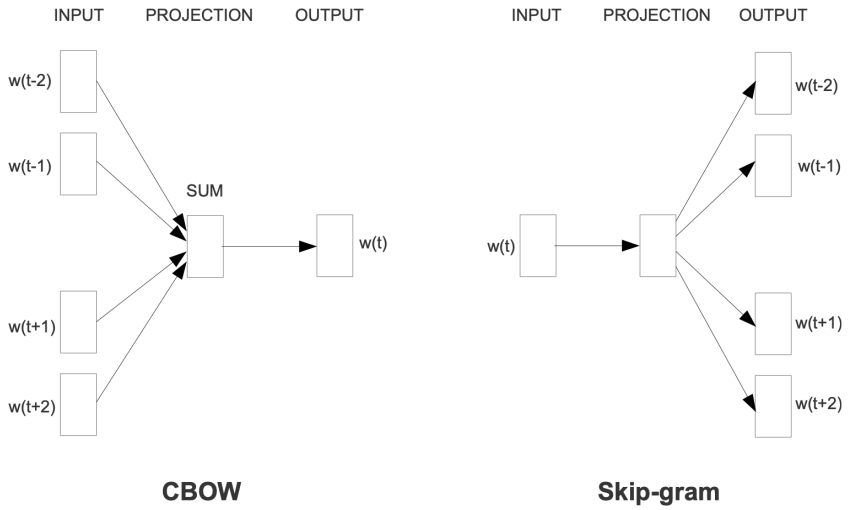
$$W_{\text{TF*IDF}(t_i,d_j)} = tf_{t_i,d_j} \times \left(1 + \log \frac{D}{df(t_i)}\right)$$

The last NLP algorithm utilized in this work is Doc2Vec, developed by Le and Mikolov in 2014 as an extension of the Word2Vec model (57). To better understand Doc2Vec, it's important to first understand Word2Vec, that is an algorithm that generates vector representations of words by considering the context in which a word appears, in other words, the meaning of a word is characterized by its surrounding words (58).

> A more modern alternative to the bag-of-words model is word2vec, an algorithm that Google released in 2013 (T. Mikolov, K. Chen, G. Corrado, and J. Dean. *Efficient Estimation of Word Representations in Vector Space*. arXiv preprint arXiv:1301.3781, 2013). The word2vec algorithm is an unsupervised learning algorithm based on neural networks that attempts to automatically learn the relationship between words. The idea behind word2vec is to put words that have similar meanings into similar clusters; via clever vector-spacing, the model can reproduce certain words using simple vector math, for example, *king – man + woman = queen*. (59)

Word2Vec comes with two types of implementation: Continuous Bag of Words (CBOW) and Skip-Gram (60). As illustrated in Figure 2.7, adapted from Mikolov et al., CBOW uses the surrounding context to predict a target word, on the other hand Skip-Gram takes a target word and predicts its surrounding context.

Figure 2.7: CBOW and Skip Gram.



Source: Adapted from Mikolov's article (61).

Mikolov et al. (61) defined the goal of Skip-Gram training as maximizing the average log probability in the following formula, where k is the size of the training window and T represents the total number of words in the training corpus:

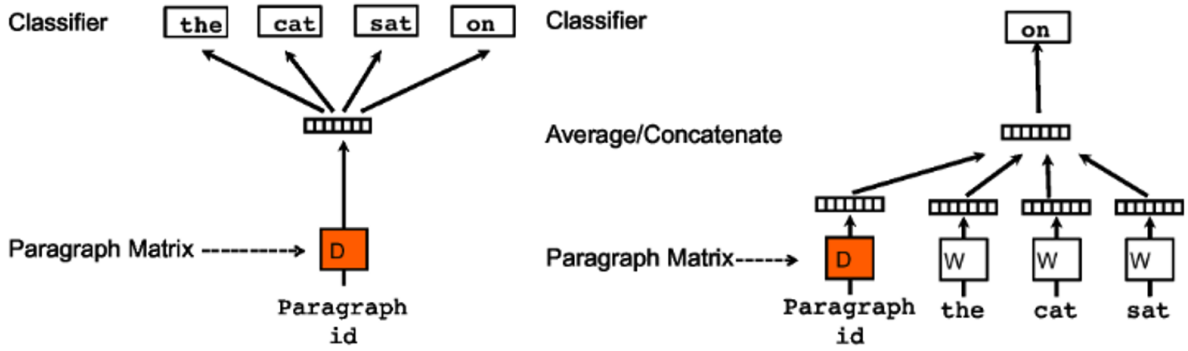
$$\frac{1}{T}\sum_{t=1}^{T}\left[\sum_{j=-k}^{k}\log p(w_{t+j} \mid w_t)\right]$$

And the probability of successfully predicting a word, given a vocabulary of size V, as:

$$p(w_i \mid w_j) = \frac{\exp(u_{w_i}^{\top} v_{w_j})}{\sum_{l=1}^{V}\exp(u_l^{\top} v_{w_j})}$$

As said before, Doc2Vec, introduced in the article *Distributed Representations of Sentences and Documents* (62), extends Word2Vec by incorporating the concept of a *Paragraph Vector*. It follows a similar methodology to Word2Vec, with the key addition of a unique vector representing each document or paragraph, as illustrated in Figure 2.8.

Figure 2.8: Illustration of the Doc2Vec architecture.



Source: Adapted from Le and Mikolov's article (62).

2.1.3 Artificial Balancing Algorithms

A ML model's effectiveness heavily depends on the quality of the data used for training. When dealing with highly imbalanced datasets, the model can become biased towards the class with more occurrences. This issue is particularly prevalent, as highlighted by Zheng et al. (63). In binary classification scenarios, such as the one addressed in this project, the class with a higher number of occurrences is referred to as the majority class, while the class with fewer occurrences is known as the minority class.

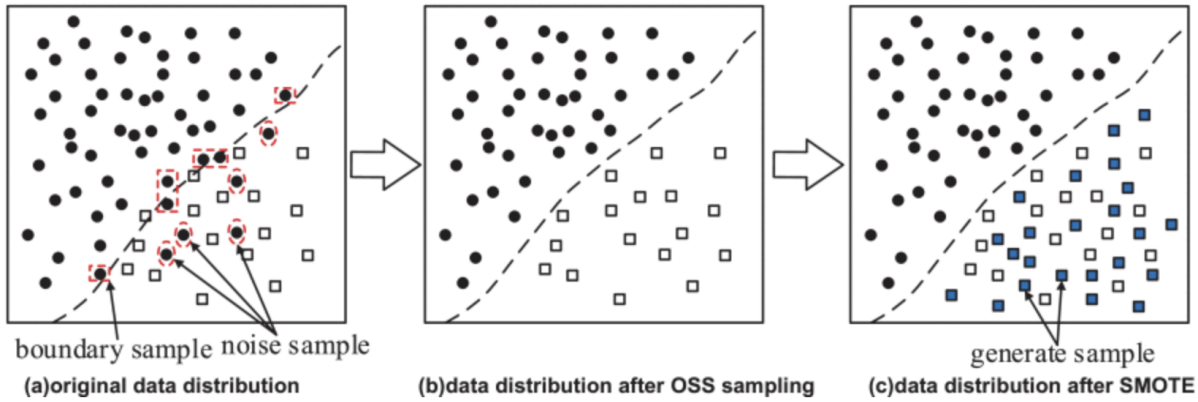
In this work, five algorithms for handling imbalanced data were evaluated: Random Undersampling (RUS), Random Oversampling (ROS), Near Miss, One-Sided Selection, and SMOTE. Each of these methods has its own advantages and limitations.

Starting with the simplest approaches, RUS and ROS, these methods are essentially opposites. RUS, as the name suggests, randomly removes samples from the majority class. This reduction in the dataset size decreases training time but risks losing valuable information. On the other hand, ROS increases the dataset size by randomly duplicating samples from the minority class. While this approach ensures better representation of the minority class, it increases the training time and may lead to overfitting (64).

The Synthetic Minority Over-sampling Technique (SMOTE), introduced by Chawla et al. in 2002 (65), offers another approach to addressing imbalanced datasets. Rather than duplicating existing minority samples, it generates synthetic data points by interpolating between a sample from the minority class and one of its k- nearest neighbors. By increasing the diversity of the minority class without simply duplicating them, SMOTE reduces the risk of overfitting when compared to ROS.

SMOTE and ROS belong to the group of algorithms that focus on altering the minority class. Differently, algorithms like RUS, Near Miss, and One-Side Selection (OSS) target the majority class. OSS works by removing samples from the majority class that are farthest from the minority class, effectively reducing noise and imbalance (66). A notable hybrid approach was proposed by Jiang et al. (67), where OSS was first used to reduce the majority class, followed by SMOTE to generate synthetic samples for the minority class, as illustrated in Figure 2.9.

Figure 2.9: Hybrid sampling process combining OSS and SMOTE.



Source: Adapted from Jiang et al. (67).

Lastly, Near Miss, like OSS, uses distances to remove samples from the majority class. However, it uses different strategies to balance the dataset. Near-Miss-1 selects majority class samples that are closest to minority class instances. Near-Miss-2 removes majority class samples that are farthest from other majority class samples but still near the minority class. Finally, Near-Miss-3 selects a fixed number of majority class samples closest to each minority class instance (68).

2.1.4 Metrics

A Confusion Matrix (CM) serves as a fundamental tool for evaluating the performance of a machine learning model (69). As shown in Table 2.2, the CM is composed of four elements: True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN).

	Predicted Positive	**Predicted Negative**
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

Table 2.2: Confusion Matrix with True Positives, True Negatives, False Positives, and False Negatives.

Harrison (70), in his book, outlines several performance metrics implemented in Python's *sklearn.metrics* module. Among these metrics are Accuracy, Precision, Recall, and F-1 Score. Harrison defines Accuracy as the percentage of correct classifications, computed using the formula:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

To compute the F-1 Score, Precision and Recall must first be defined. As explained by Cook (71) and Harrison (70), Precision represents the proportion of true positives among all predicted positives and is given by:

$$\text{Precision} = \frac{TP}{TP + FP}$$

Recall, also known as Sensitivity, quantifies the proportion of actual positives correctly identified by the model and is calculated as:

$$\text{Recall} = \frac{TP}{TP + FN}$$

The F-1 Score (70) is calculated as the harmonic mean of Precision and Recall, providing a balanced metric that considers both false positives and false negatives, being defined as:

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Finally, in this work *Matthews Correlation Coefficient* (MCC) was used as well. MCC is a balanced evaluation metric that takes into account true and false positives, as well as true and false negatives, making it suitable even for imbalanced datasets (72):

$$\text{MCC} = \frac{(TP \cdot TN) - (FP \cdot FN)}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

> MCC - It is essentially a correlation coefficient between the true and predicted classes and achieves a high value only if the classifier obtains good results in all entries of the confusion matrix. The Matthews correlation coefficient is bounded to [-1, +1], where a value of 1 represents a perfect prediction, 0 random guessing, and -1 total disagreement between prediction and observation. (73)

Chicco and Jurman (74) (75) emphasized a significant limitation in using accuracy as an evaluation metric for machine learning models in binary classification tasks involving imbalanced datasets. In such scenarios, a model that consistently predicts the majority class can achieve a high accuracy rate, even though it fails to effectively distinguish between classes. Metrics like the MCC and F1-Score offer more robust alternatives. While MCC accounts for all entries in the confusion matrix, providing a comprehensive measure of model performance, the F1-Score remains one of the most widely used metrics in machine learning, applicable to both binary and multi-class classification tasks.

2.1.5 Grid Search

All ML models have hyperparameters, and their optimal performance can only be achieved after being tuned properly (76). One of the simplest and most commonly used methods for identifying the best hyperparameter configuration is Grid Search (77). Grid Search systematically evaluates a list of predefined hyperparameter values, testing every possible combination, and returns the configuration that achieves the best performance, this could be achieved using Python's scikit-learn package (78).

As this technique exhaustively evaluates all possible combinations of hyperparameters, it can be com-

putationally intensive and time-consuming. Moreover, since the user must define the range and list of hyperparameters to test, there is a risk of overlooking potential optimal values outside the specified list.

2.2 RELATED WORK

The vast majority of machine learning work in the cybersecurity field is focused on the area of malware detection. For example, Chumachenko (79) explored several Machine Learning algorithms for the purpose of detecting and classifying malware. His research concluded that, with his dataset, the most accurate algorithms were Random Forest, followed by J48. However, Chumachenko struggled with a huge number of features, 70518 in total, estimating that it'll take around 23 hours just to load the dataset. So some filtering was implemented to achieve the amount of 306 features, for this all features were ranked and then merged into new subsets to be selected again.

Furthermore, Rathore (80) compared the results between the use of Machine Learning and Deep Learning in malware detection, concluding that Random Forest outperformed Deep Neural Networks. In both studies, Random Forest proved to be a promising option for threat detection.

Finder et al. (81) developed a framework called *time-interval-based active learning*, which leverages time intervals to detect emerging malware trends. Several machine learning classifiers were evaluated, with the Support Vector Machine (SVM) showing the best performance in this particular study.

As Rathore (80), Ding (4) implemented Deep Learning techniques for detecting LOLBins, achieving an the accuracy of 99.45%. In terms of feature extraction, Ding explored several text vectorization methods, including BOW, TF-IDF, Word2Vec, Doc2Vec, and WV-TI (Word2Vec combined with TF-IDF). Ultimately, Word2Vec was selected as the most effective technique. In addition to text vectorization, Ding introduced additional features based on the presence of network commands, binary files, the -c"option, and installation commands. This approach was named **LOLWTC**, which stands for "*LotL Attack Detection Method Based on Word2Vec and TextCNN*".

Ding cited the work of Boros (82), who also developed labels for binaries, paths, parameters, networking, and other recurring patterns in the LOLBins context. Boros identified specific strings frequently appearing in LOLBins scenarios and created regular expression (regex) patterns to detect and replace them.

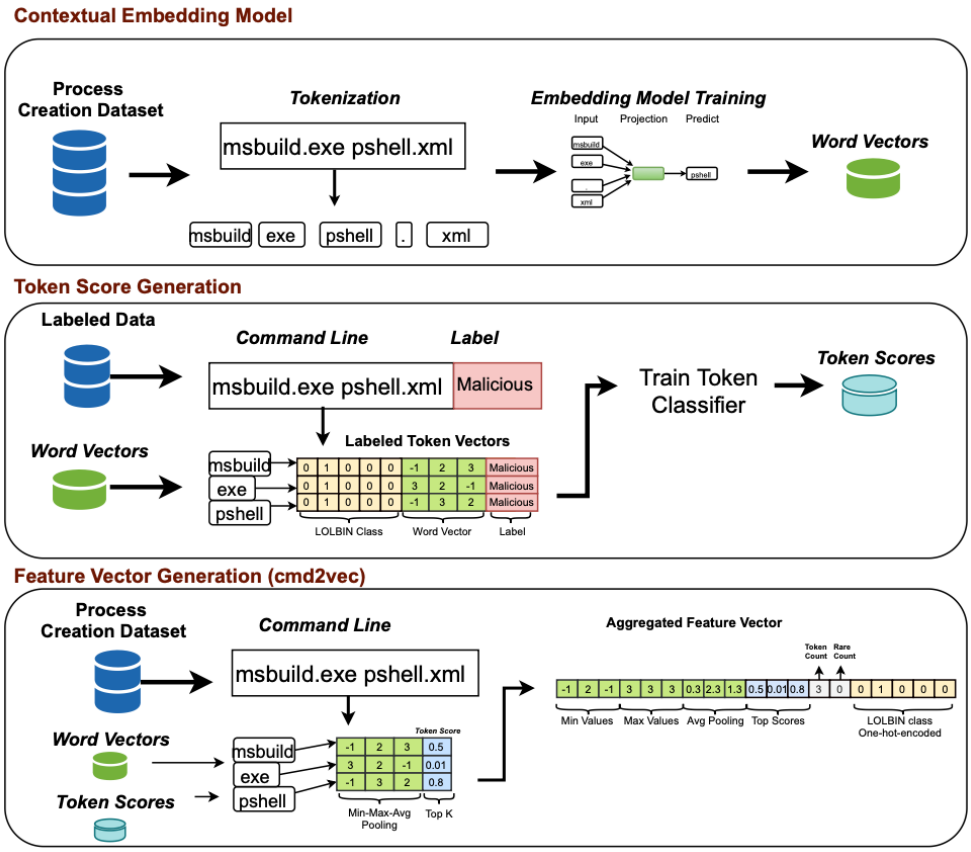
> Using a list of known LOLBins malicious looking commands we build a series of regexes where purpose is to identify LOLBins components in commands. For example, let's take the following command:
>
> ```
> python-c'import sys,socket,os,pty;s=socket.socket();s.connect((os.getenv(""RHOST""),int(os.getenv(""RPORT""))));[os.dup2(s.fileno(),fd)for fd in(0,1,2)];pty.spawn(""/bin/sh"")'
> ```
>
> There are a couple of pieces of information we can extract using regex. We can detect the use of the pty library `r'python.*c.*pty'` or we can detect the socket library `r'python.*c.*socket'`, the connection itself `r'python.*c.*connect'` or the shell invocation itself `r'python.*c.  *pty.*sh'`. The regexes can be more or less permissive but the scope is to generate as much context as possible. (82)

However, rather than generating new features based on these labels, Boros used the regex replacements directly in the machine learning model, aiming to enhance the training process by improving the model's ability to recognize and handle these common patterns.

For the command in the last example, the feature extraction phase would generate tags as: `PATH_/BIN/SH, COMMAND_PYTHON, COMMAND_FOR, KEYWORD_-C, KEYWORD_SOCKET, KEYWORD_OS, KEYWORD _PTY , KEYWORD_PTY.SPAWN, python_socket, python_shell, import_pty`.

Most of the works that aimed at detecting LOLBins in general, in which was included other systems such as Windows, for example, addressed this interaction between different ways of representing texts originating from command lines. An example of this was Ogun (83) who implemented a way of assigning features that he called *cmd2vec*. In his work, Ongun (83) submitted the tokens for each command to the Doc2Vec or FastText algorithm for vectorization, as seen in Figure 2.10. No Code was shared by Ongun, however, based on the description of the paper a PoC (Proof of Concept) was created.

Figure 2.10: Cmd2Vec - Proposed by Ongun



Source: Adapted from Ongun's work (83)

Firstly, Ongun created a "*Contextual Embedding Model*". In this phase, each input is splitted in tokens, where I choose, as seen in Code 2.1, to simply split by spaces. Then, these tokens are converted to vectors by a NLP algorithm, in this case or Word2Vec or FastText.

```
lista = []
for i in Dados:
    print(i.split())
    lista.append(i.split())
from gensim.models import Word2Vec, FastText
```

```
word2vec_model = Word2Vec(sentences=lista, vector_size=100, window=5, min_count=1,
    sg=0)
fasttext_model = FastText(sentences=lista, vector_size=100, window=5, min_count=1,
    sg=0)
vector_w2v = word2vec_model.wv['git']
vector_ft = fasttext_model.wv['git']
```

Code 2.1: Contextual Embedding Model

Secondly, a score is created for each token, because Ongun believed that malicious commands tend to rare words or parameters than the usual common commands. To explore this, he trained his first database using the Random Forest algorithm and calculates the average probability that each token appears on a leaf of the tree with a final label representing the LOLBins commands.

In Code 2.2, I replicate Ongun's approach by combining Word2Vec embeddings and a Random Forest classifier to evaluate the importance of tokens in a binary classification task. After training, the model associated each token with the leaf nodes it reaches in the Random Forest. For tokens labeled as 1, the probabilities of belonging to this positive class are aggregated and normalized based on the number of samples in each leaf. The result is the average probability of each token being classified as 1, which provides insight into how significant each token is for the classification task based on its path through the model's decision trees.

```
from sklearn.ensemble import RandomForestClassifier
from gensim.models import Word2Vec
import numpy as np
labels = [1, 0, 1, 0, 1]
word_vectors = word2vec_model.wv
entry_vectors = []
entry_labels = []
for i, tokens in enumerate(lista_de_tokens):
    for token in tokens:
        if token in word_vectors:
            entry_vectors.append(word_vectors[token])
            entry_labels.append(labels[i])
random_forest_model = RandomForestClassifier(n_estimators=100, random_state=42)
random_forest_model.fit(entry_vectors, entry_labels)
num_samples_per_leaf = random_forest_model.apply(entry_vectors)
token_leaf_probabilities_sum = {}
for i, tokens in enumerate(lista_de_tokens):
    if labels[i] == 1:
        for j, token in enumerate(tokens):
            if token not in token_leaf_probabilities_sum:
                token_leaf_probabilities_sum[token] = []
    token_leaf_probabilities_sum[token].append(random_forest_model.predict_proba([
        entry_vectors[i]])[0][1] / num_samples_per_leaf[i, j])
mean_leaf_probabilities_label_1_normalized = {token: np.mean(probs) for token,
    probs in token_leaf_probabilities_sum.items()}
```

Code 2.2: Token Score Generation

Finally, Ongun developed a new dataset creation process through a function called "*CMD2VEC*"during the phase called "Feature Vector Generation". This step generates a new dataset by calculating various statistical measures from the word vectors and the probabilities gathered in previous steps. For each entry, the function extracts the top three minimum, maximum, and average values from the word vectors associated with each token. Additionally, it captures the top three leaf scores derived from the Random Forest training to reflect the importance of specific tokens for classification. Finally, he also included a count of the entry tokens and the rarest ones, as he noticed that in his database the binaries used in attacks sometimes appeared only a few times. Each processed entry results in a feature vector that contains these statistical metrics, along with a label for supervised learning.

```
for i, tokens in enumerate(lista_de_tokens):
    word_vecs = [word_vectors[token] for token in tokens if token in word_vectors.key_to_index]
    if len(word_vecs) > 0:
        word_vecs = np.array(word_vecs)
        if len(word_vecs) >= 3:
            min_values = np.min(word_vecs, axis=0)[:3]
            max_values = np.max(word_vecs, axis=0)[:3]
            avg_values = np.mean(word_vecs, axis=0)[:3]
        else:
            min_values = np.zeros(3)
            max_values = np.zeros(3)
            avg_values = np.zeros(3)
        token_scores = [mean_leaf_probabilities_label_1_normalized[token] for token in tokens if token in mean_leaf_probabilities_label_1_normalized]
        token_scores.sort(reverse=True)
        max_scores = token_scores[:3]
        num_tokens = len(tokens)
        rare_count = sum(1 for token in tokens if token not in word_vectors.key_to_index or word_vectors.get_vecattr(token, 'count') <= 1)
        label = labels[i]
        entry_data = np.array((min_values[0] if len(min_values) > 0 else 0, min_values[1] if len(min_values) > 1 else 0, min_values[2] if len(min_values) > 2 else 0,
                               max_values[0] if len(max_values) > 0 else 0, max_values[1] if len(max_values) > 1 else 0, max_values[2] if len(max_values) > 2 else 0,
                               avg_values[0] if len(avg_values) > 0 else 0, avg_values[1] if len(avg_values) > 1 else 0, avg_values[2] if len(avg_values) > 2 else 0,
                               max_scores[0] if len(max_scores) > 0 else 0, max_scores[1] if len(max_scores) > 1 else 0, max_scores[2] if len(max_scores) > 2 else 0,
                               num_tokens, rare_count, label),
                              dtype=[('min_val_0', np.float64),('min_val_1', np.float64),('min_val_2', np.float64),
                                     ('max_val_0', np.float64),
                                     ('max_val_1', np.float64),
                                     ('max_val_2', np.float64),
```

```
                                        ('avg_val_0', np.float64),
                                        ('avg_val_1', np.float64),
                                        ('avg_val_2', np.float64),
                                        ('max_score_0', np.float64),
                                        ('max_score_1', np.float64),
                                        ('max_score_2', np.float64),
                                        ('num_tokens', np.int64),
                                        ('rare_count', np.int64),
                                        ('label', np.int64)])
            new_dataset.append(entry_data)
    new_dataset = np.array(new_dataset)
```

Code 2.3: Feature Vector Generation

This structured data enables the model to better differentiate between command sequences, especially focusing on detecting obfuscated or malicious patterns based on their token-level attributes and their traversal paths through the decision trees. With this approach, Ongun managed to achieve the F1 Score of 0.96 using FastText and 0.94 with Doc2Vec.

Unfortunately, Ongun did not provide the dataset used in the tests to evaluate the PoC created.

Still in the field of LOLBins detection, Alaa AbuShqeir (1) applied a binary classification approach using a dataset composed of 350 legitimate and 300 malicious commands, primarily targeting Windows machines. The malicious LOLBins dataset was sourced from the LOLBAS project, similar to the GTFOBins repository (2) but focused only on Windows binaries. AbuShqeir enhanced the preprocessing phase by creating specialized regex patterns to tokenize the commands more effectively. This approach replaced most of the original data with newly generated tokens. At the end, according to the author, the efficiency in detecting malicious input had a detection rate of up to 98.8% and false-positive at 4.9%.

Demmer (84) also utilized the LOLBAS project, but focused on detecting LOLBins using the SIGMA rule framework (85). The SIGMA project provides a flexible and platform-agnostic format for creating detection rules that can be converted into queries compatible with various Security Information and Event Management (SIEM) systems. Demmer developed and tested multiple custom SIGMA rules to identify specific LOLBins functions executed by common Windows binaries.

Although some works did not specifically focus on detecting LOLBins, they provided valuable insights into Machine Learning and Natural Language Processing (NLP) techniques. This was the case with the work of Hendler et al. (86), which concentrated on detecting malicious PowerShell commands. The paper utilized a dataset comprising 66,388 PowerShell commands, of which 6,290 were malicious. An additional set of 471 commands, provided by Microsoft Security Experts, was used for evaluation. One of the challenges identified was that PowerShell commands are often heavily obfuscated, making detection more difficult. However, the study demonstrated that deep learning approaches, particularly character-level Convolutional Neural Networks (CNNs), are highly effective in detecting obfuscation techniques such as base64 encoding and dynamically generated commands.

2.2.1 Classification of Linux Commands in SSH Session by Risk Levels

A significant challenge with previous works is the lack of publicly available datasets, which hinders the ability to reproduce and extend the research. Additionally, the code used in these studies is often not provided. However, in the case of Ongun (83), the methodology was reproducible. In a machine learning context, though, the dataset remains the foundation of model training, making its availability crucial for further advancements. For this purpose, the work of Thuy Ngan (13) based our project.

In Ngan (13) master's thesis, it aimed to classify Unix commands based on risk levels. To achieve this, it were collected 660 .bash_history files from GitHub and used logs from two honeypots obtained from another study conducted at the same university.

The total corpus comprised 905,405 commands. In an initial analysis, Ngan performed a binary classification of the data, assuming that commands from GitHub were legitimate, while those from the honeypots were considered malicious. In a second phase, central to the study's goal, all data was processed using a labeling function that categorized each command by risk level.

After training the machine learning algorithms, the predictions were evaluated using a confusion matrix, where both false positives and false negatives were measured. False negatives were particularly critical, as misclassifying a malicious command could lead to significant impacts on a system.

The data, sourced from two different origins, needed to be cleaned and normalized before being processed by text algorithms such as Bag of Words and Doc2Vec.

Before applying NLP text representation techniques, the data underwent a preprocessing phase. During this stage, information like emails, URLs, IP addresses, file paths, and text were replaced with appropriate textual representations. To perform this task, the Python library Textacy (87) was used, which provides specialized functionalities for NLP. For additional preprocessing tasks, custom functions were created using regular expressions to replace specific data with designated strings.

The .bash_history files were fragmented into records corresponding to each collected session. As a result, it was necessary to read each file, with every command line being processed through a normalization function. This function applied various text-cleaning operations, and finally, all commands containing *&&* or *;* were split into separate commands.

Similarly, the commands executed in the honeypots were stored in JSON files for each session, accessible via the *cowrie_command_input* key. These commands were extracted and subjected to the same normalization functions, ensuring consistent treatment of the commands, as illustrated in the example in Code 2.4.

```
1 git commit --am _STRING_
2 echo at http:_PATH_ my email is _EMAIL_
3 ping _IP_ && ping google.com
4 rm _PATH_  |  rm _PATH_ & cp _PATH_ _PATH_
```

Code 2.4: Ngan's Sample

Ngan (13) initially conducted a binary classification scenario, where commands from GitHub were

labeled as legitimate, while those originating from honeypots were considered malicious.

In addition to this approach, the scenarios were segmented based on the presence or absence of context. Given the lack of additional information about command execution in Linux history files, which only store the commands themselves, the context of a command was defined as the two preceding commands, if available.

The scenarios were further subdivided according to the NLP representation methods used, such as Bag of Words, TF-IDF, and Doc2Vec. After training the scenarios with instance-based learning algorithms like K-Nearest Neighbors (KNN) and Support Vector Machines (SVM), Ngan achieved the results shown in Table 2.3:

Context	Representation	Classification	Accuracy	False Neg
1-Command	1-gram + BoW	KNN	89.13%	28,878
3-Command	1-gram + TF-IDF	Linear SVC	98.13%	2,292
	3-gram + TF-IDF		98.27%	2,066
	3-gram + Doc2Vec		96.71%	1,499

Table 2.3: Results obtained by Ngan in the binary classification task

From the implementation using TF-IDF, Ngan introduced the concept of command context or command window. Due to the lack of additional data, since the .bash_history file only stores the commands themselves — omitting information such as user, process, and execution time — the context of a command was defined as the two preceding commands, if available. To achieve this, Ngan created a function that traverses each command, appending the two preceding commands and saving the result in a new list, as shown in Code 2.5. In addition to creating the 3-command window, this new list was processed using TF-IDF in two ways: with Unigram and Trigram models. As explained by Jurafsky and Martin (88), an N-gram is a sequence of N words, where the sequence is treated as a single token.

```
def create_n_command_by_sliding_window(session_cmds,
labels, window_size=3):
    cmds_flat = []
    labels_flat = []

    for sess_idx, cmds in enumerate(session_cmds):
        for i in range(len(cmds) + 1):
            start_idx = max(0, i - window_size)
            if start_idx == i:
                continue
            cmds_flat.append(" ".join(cmds[start_idx:i]))
            labels_flat.append(labels[sess_idx])
return cmds_flat, labels_flat
```

Code 2.5: Ngan's Window of Commands

Doc2Vec was developed by Le and Mikolov as a simple extension to Word2Vec. As explained by Lau and Baldwin (57), the purpose of this extension was to create vector representations for entire documents

rather than individual words. Ngan once again used the 3-command window for processing with Doc2Vec, achieving an accuracy rate of 98.23% with only 707 false positives. This was the best scenario achieved for the binary classification.

However, the core of Ngan's project was to classify by risk levels the Linux commands submitted to honeypots, allowing for improved management of permissions for agents interacting with the system. This classification would enable a more precise evaluation of executed commands, giving the honeypots the ability to appropriately manage the level of freedom granted to these agents during their interaction with the system.

The manual labeling was unfeasible given the large number of entries, 905,045 commands. Therefore, the Snorkel library was used to automate the labeling process. Snorkel allows the creation of datasets using heuristic rules, as described on its development site (89). Functions were created to generate the new labels and used to categorize the commands based on their usage type, such as those related to user information, system, hardware, and processes. Additionally, manual functions were created to capture other characteristics, such as command length, base64 content, packet manipulation, and service interactions, among others, as demonstrated in the code 2.6, with the classes UNKNOWN and R [0-4].

```
labeling_functions = [
    make_keyword_lf(f_name=f_name, keywords=keywords, label=label)
    for f_name, (label, keywords) in keyword_labeling_func_config.items()
]
labeling_functions += [ lf_long_cmd,
                        lf_base64,
                        lf_history,
                        lf_install,
                        lf_schedue,
                        lf_firewall,
                        lf_disable_services,
                        lf_sensitive_keywords,
                        lf_execute ]
```

Code 2.6: Ngan's Window of Commands

The data were processed by Ngan using both Bag of Words and Doc2Vec, each with the Logistic Regression algorithm. The Doc2Vec approach achieved an accuracy rate of 99.58%, as shown in Table 2.4

Representation	Classification	Accuracy
Count-Vector	Logistic Regression	95.05%
Doc2Vec		99.58%

Table 2.4: Risk Classification

With the exception of the BoW implementation using KNN, Ngan's other implementations (13) achieved good results in terms of accuracy and false negative counts. However, upon reviewing Ngan's work, we extended the study by implementing additional algorithms in the same scenarios. As a result, we obtained higher accuracy across all scenarios using neural networks, as shown in Table 2.5.

Algorithm	NLP	Accuracy	Execution Time	False Neg.
Neural Network	BoW	93.13%	4 min 19.6 sec	5,288
Neural Network	TF-IDF	99.49%	16 min 2.7 sec	775
Neural Network	Doc2Vec	99.23%	197 min 37.4 sec	792

Table 2.5: Best Results Obtained with Neural Networks

The application of other machine learning algorithms to the same scenarios highlights the complexity of selecting the right algorithm. For models designed to be implemented in intrusion detection systems, the choice of algorithm must go beyond simply maximizing accuracy. It is essential to strike a balance between accuracy, false negative counts, and the time required for training and prediction.

2.2.2 Command Obfuscation

LOLBins are inherently stealthy techniques; however, some commonly used LOLBins commands are already recognized by antivirus solutions and can be flagged by detection systems.

To evade detection, adversaries often apply command obfuscation techniques, which transform commands syntactically without changing their underlying behavior. These transformations hinder pattern recognition by both signature-based and machine learning-based detection tools.

For the purpose of this work, we define:

- **Plain commands**: Legitimate commands executed without any alterations or obfuscation techniques. These commands maintain their original syntax and are generally human-readable.
- **Obfuscated commands**: Commands that have been transformed using one or more obfuscation techniques to evade detection, while still preserving their intended execution logic.

Tsai et al. (90) conducted a study focusing on the classification of malicious and obfuscated PowerShell commands, demonstrating how these techniques significantly increase the complexity of detection.

Three categories of obfuscation are mentioned, compression, manipulation of *string* and *enconding schemes*, along side the *Helper Functions*. *Enconding schemes* are methods of displaying data in a specific format such as Base64, Hexadecimal, Unicode, among others. *Helper Functions* are defined as functions that are created to be combined with other obfuscation methods, performing functions such as text splitting and replacement, as well as *Randomcase* where each character is converted using a pattern randomly set. All the methods studied by Tsai are defined in Table 2.6. As a solution to the problem of malicious obfuscated *PowerShell* commands, Tsai created the *framework* which he named *PowerDP* (90), which performs the deobfuscation and subsequent static classification of PowerShell commands.

The PowerDP framework operates in two phases: *De-Obfuscating* and *Profiling* PowerShell commands. In the de-obfuscation phase, PowerDP uses multi-label classification to identify the various obfuscation techniques present in the commands. This is essential because attackers often employ multiple layers of obfuscation to evade detection. The classifier achieves an impressive obfuscation detection accu-

Priority	Obfuscation method	Category
1	Tick	String manipulation
2	Concatenation	String manipulation
3	Reversing	String manipulation
4	Reordering	String manipulation
5	Replacement	Helper function
6	Splitting	Helper function
7	ASCII encoding	Encoding scheme
8	Binary encoding	Encoding scheme
9	Hexadecimal encoding	Encoding scheme
10	Octal encoding	Encoding scheme
11	Binary XOR encoding	Encoding scheme
12	Compression	Compression
13	SecureString encoding	Encoding scheme
14	Base64 encoding	Encoding scheme
15	Randomcase	Helper function

Table 2.6: Obfuscation methods and their categories

racy of 99.82%.

Once the obfuscation techniques were identified, PowerDP applied a de-obfuscation process using regular expressions and static replacements. This process successfully recovers the original command content with an accuracy of 98.11% across 15 different obfuscation techniques. This concludes their first phase.

In the second phase, after the PowerShell commands are de-obfuscated, the framework moved to behavioral profiling, where it analyzes the intent behind the commands. This is accomplished by extracting features from the abstract syntax tree (AST) of the commands. The framework then classifies the commands into various behavioral categories, such as code injection, malware download, or system reconnaissance. This behavioral profiling phase achieved an accuracy of 98.53% in identifying malicious activities.

Other works had focused on detecting obfuscated malicious anomalies, however most of them have focused on detecting *malware* rather than command lines. Kolli et al. (91), for example, applied Artificial Neural Networks to classify 280 samples of *metamorphic malware*. These type viruses change their structure during code execution, bypassing some detection software.

Kolli et al. (91) developed a neural network model designed to detect obfuscated malware by analyzing underlying patterns using similarity-based techniques. The paper highlights the critical importance of minimizing both false positives and false negatives, which are persistent challenges in malware detection. Traditional methods often produce a high number of false alarms, overwhelming security analysts and reducing system efficiency. The proposed neural network approach demonstrated a significant reduction in false positive rates, decreasing to just 8% as the size of the training dataset increased, showcasing the model's effectiveness in improving detection accuracy.

Although false positives present a challenge, false negatives are far more critical, as they indicate that the model failed to detect malicious code or malware. This could harm the system without raising any

alarms.

Khan (92) tried to detect obfuscated malware using an Artificial Neural Network (ANN) and the CIC-MalMem-2022 memory analysis dataset. Khan achieved a accuracy rate of 99.72%. Similar to Khan, Hasan et al (93) proposed a cost-effective solution that leverages memory dump analysis combined with various machine learning algorithms, such as decision trees, ensemble methods, and neural networks. Their research was based on an dataset of 58,596 entries, with 55 features binary classified in malicous or not. Their results demonstrated that the proposed framework was effective in detecting obfuscated malware across multiple categories, including spyware, ransomware, and Trojan horses.

Hasan et al. (93) faced the challenge of working with a highly imbalanced dataset. To address this issue, they employed two approaches: undersampling the majority class and oversampling the minority class. For reducing the majority class, they used the Random Undersampling (RUS) and Near Miss algorithms. Random Undersampling (RUS) reduces the size of the majority class by randomly discarding samples, while Random Oversampling (ROS) increases the size of the minority class by randomly duplicating samples. As noted by Weiss (94), these techniques have limitations: undersampling can result in the loss of valuable information, while oversampling risks overfitting by duplicating data. Near Miss, on the other hand, selectively removes samples from the majority class based on their proximity to the minority class, aiming to retain more useful information. To increase the size of the minority class, Hasan et al. employed ADASYN (Adaptive Synthetic Sampling), which outperformed other methods in balancing the dataset.

3 METHODOLOGY

This chapter outlines the methodology used in this thesis, divided into two main parts. The first part focuses on the classification of plain LOLBins in Linux, detailing the process from dataset construction to identifying the optimal combination of ML, NLP, and data balancing algorithms.

The second part addresses the classification of obfuscated LOLBins in Linux. Here, the previously constructed dataset is increased with obfuscated versions of the commands. Subsequently, the optimal model identified in the first part is fine-tuned to effectively classify the obfuscated commands.

Before proceeding, it is crucial to address the environment configuration, as it directly impacts performance. A more powerful system can train models with more features and significantly reduce the time required for training. The system configuration used in this work is detailed in Table 3.1.

Component	**Details**
CPU	Apple M2 Max
Memory (RAM)	96 GB
GPU	Apple M2 Max with 38 Cores
Operating System	macOS 15.1.1 (24B91)
Python Version	3.11.5 (Anaconda Distribution)
Key Python Libraries	scikit-learn (1.2.2) TensorFlow (2.15.1) PyTorch (2.0.1) NumPy (1.24.3) Pandas (2.1.1) Matplotlib (3.7.2)

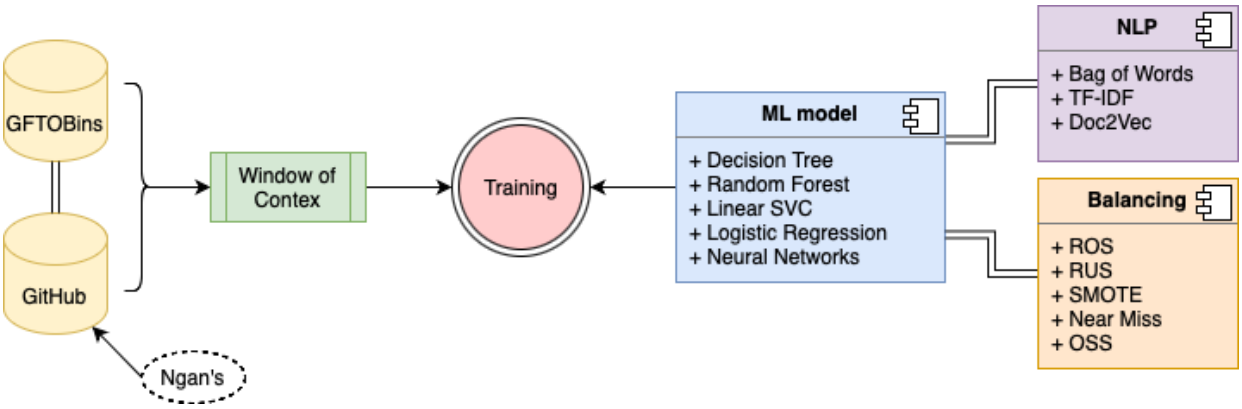
Table 3.1: System Environment for Experimentation

3.1 CLASSIFICATION OF PLAIN LOLBINS

The first phase of this study aims to identify the optimal combination of machine learning (ML) models, natural language processing (NLP) techniques, and data balancing strategies, as shown in Figure 3.1.

This research builds directly on the work of Ngan (13), which provided both the dataset and the code for the initial data normalization process. While Ngan's work primarily focused on preprocessing and preliminary classification, this study expands upon that foundation. It explores alternative normalization strategies, refines feature engineering, and evaluates a wider range of ML models and balancing techniques to enhance detection performance.

Figure 3.1: Overview of the first test.



Source: Author

This section is structured into three parts: *Data Normalization and Tokenization*, *Creation of the Dataset*, and *Tests Order*.

3.1.1 Data Normalization and Tokenization

Before delving into dataset creation, it is essential to address the data normalization process. Most studies focused on detecting malicious command lines—regardless of the operating system—have sought to improve the normalization of input text before it is transformed into features, as discussed in Section 2.2. For example, Ongun et al. (83) developed a dedicated function to normalize text and generate feature representations for their dataset.

Text preprocessing is crucial in natural language processing (NLP) pipelines. One of the key steps in this phase is **tokenization**, which refers to segmenting a string of text into smaller, meaningful units called *tokens*. These tokens can represent words, subwords, or characters, depending on the analysis granularity required by the task (95).

Tokenization is not limited to simply splitting a string by whitespace; it often involves removing or handling punctuation marks, memorable characters, and irrelevant elements, such as stopwords, to enhance the quality and consistency of the textual data (38). For command-line analysis, tokenization may also need to account for syntactic elements such as flags, paths, and variables, which carry semantic weight in this context.

The preprocessing approach, in this work, followed a similar approach to Ngan's. For instance, Ngan's preprocessing method indiscriminately suppressed complete paths in commands, transforming strings such as "*/bin/bash -p*"into "*_PATH_ -p*". While this approach generalizes file paths, it could result in the loss of critical information, such as the command being executed. Additionally, the frequent usage of environment variables in LOLBins presented another challenge. Since variable names can vary, models might incorrectly interpret different variable names as representing different actions, even though they signify the same operation.

```python
def replace_file_path(text):
    return re.sub(r"((?<= )[^ ]*/)([^/ ]*)", r"_PATH_/\2", text)
def replace_env_variables(text):
    text = re.sub(r"\$([A-Za-z_][A-Za-z0-9_]*)", r"$_ENV", text)
    text = re.sub(r"([A-Za-z_][A-Za-z0-9_]*)=", r"_ENV_=", text)
    return text
```

Code 3.1: Python script for text manipulation

To address these issues, modifications were made to the preprocessing functions. Specifically, the *replace_file_path* function was updated to replace only the directory portion of a path with the string _PATH_, while retaining the file name at the end of the path. Similarly, the *replace_env_variables* function was adjusted to normalize variable declarations by replacing variables declarations and uses with _ENV and $_ENV, as seen in Code 3.1.

As will be showed further on, this revised approach provided a foundation for the first normalization function constructed in this work, which builds upon Ngan's normalization technique despite this two changes, and it can be found on Attachment I.2.

3.1.2 Creation of the Dataset

As mentioned in Section 2.2.1, one of the challenges of this project was finding a proper dataset to serve as a basis for our research. Fortunately, Ngan's master's thesis (13) pursued a similar line of work by classifying malicious Linux commands, and provided all the code and datasets through Kaggle[1].

Ngan used a dataset comprising two parts: one sourced from GitHub, where all commands were considered benign, and another obtained from two honeypots, where all commands were deemed malicious. For this work, we chose to use the benign portion of Ngan's dataset. For the malicious portion, we relied on the GTFOBins repository (2), which provided its dataset in JSON format[2].

```python
gtfo_df = pd.read_json('https://gtfobins.github.io/gtfobins.json')
gtfo_list = []
for function in gtfo_df.T.functions:
    for keys in function:
        for i in function[keys]:
            for chave in i:
                if chave == 'code':
                    gtfo_list.append(i[chave])
```

Code 3.2: GTFOBins's JSON to dataframe

The JSON file is processed using the Pandas module to convert it into a DataFrame, as shown in Code 3.2. This DataFrame is then normalized using the function described in Section 3.1.1, as illustrated in Code 3.3. Following this process, the LOLBins portion of the dataset consists of 1,670 unique commands and 3,463 in total.

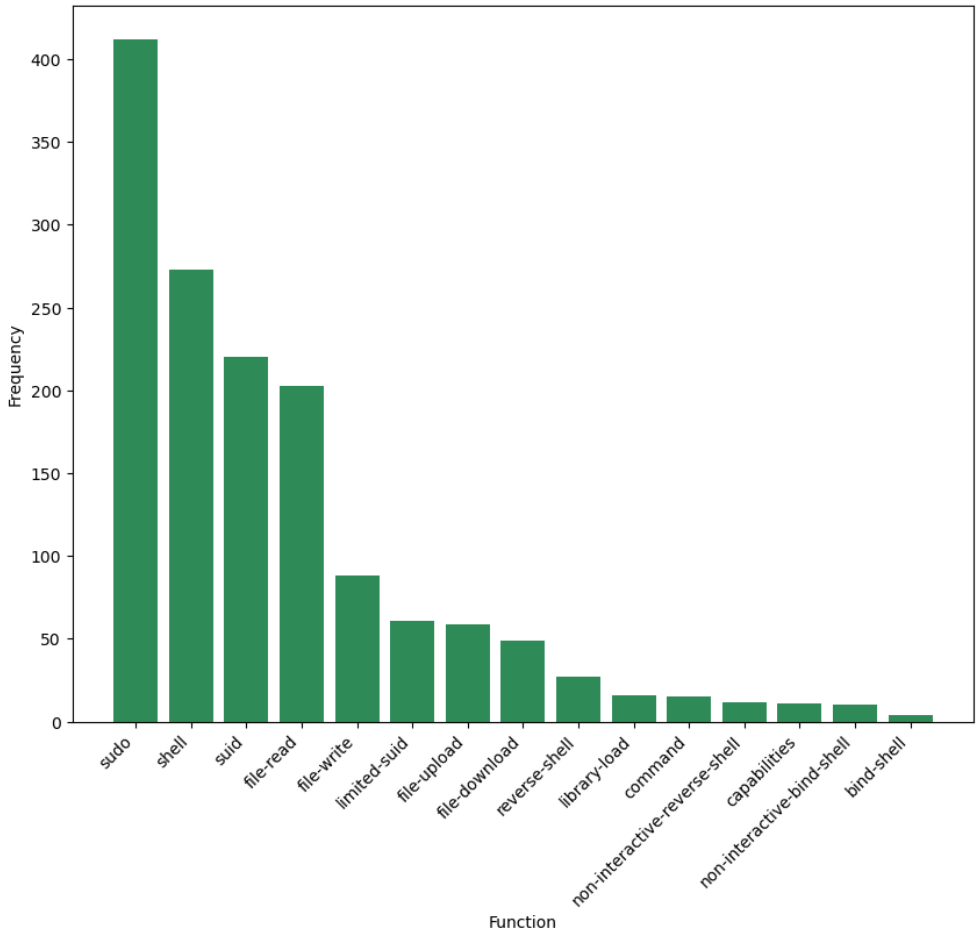
[1] <https://www.kaggle.com/datasets/thuyngandao/bashlogs>
[2] <https://gtfobins.github.io/gtfobins.json>

```
gfto_split = []
for x in gtfo_list:
    gfto_split.append(x.split('\n'))
gtfo_norm = []
for x in gfto_split:
    a = []
    for y in x:
        a.append (normalize(y))
    gtfo_norm.append(a)
for cmd in gtfo_norm:
    for x in cmd:
        if x == '':
            cmd.remove(x)
```

Code 3.3: GTFOBins's normalization

The GTFOBins repository categorizes commands into various functions based on their use cases. In Figure 3.2, it is evident that the majority of commands fall under the *sudo* category, followed by *shell* in second place and *suid* in third. The dominance of the *sudo* and *suid* categories highlights that many of these commands require specific privileges to execute, making them particularly potent in privilege escalation scenarios. Meanwhile, the *shell* category primarily focuses on spawning shells, emphasizing its utility for interactive access and further exploitation.

Figure 3.2: Frequency distribution of LOLBins functions.



Source: Author

For the benign portion of the dataset, the reading and normalization process followed the exact same implementation provided in Ngan's work, as illustrated in Code 3.4. Following this process, the benign portion of the dataset consists of 53,091 unique commands from the total of 210,402 entries.

```
data_path = "./Datasets/"
bash_logs_path = data_path + "bash_logs/"
bash_file_names = os.listdir(bash_logs_path)
benign_logs = []
for file in bash_file_names:
    with open(os.path.join(bash_logs_path, file)) as f:
        benign_logs.append([normalize(line) for line in f.read()
                            .replace("&&", "\n")
                            .replace(";", "\n")
                            .splitlines()])
benign_unique = set(itertools.chain.from_iterable(benign_logs))
while("" in benign_unique):
    benign_unique.remove("")
```

Code 3.4: Ngan's portion normalization

Upon comparing the two datasets, it was observed that the LOLBins portion constitutes approximately 1.6% of the entire dataset and 3.04% when considering only unique commands, highlighting a clearly imbalanced dataset.

An important aspect to address is the decision to use the entire corpus rather than only unique commands. In Ngan's work, one of the premises was that to better analyze LOLBin attacks, it is essential to consider additional factors such as the user, timestamp, and the commands executed together, in other words, the context in which the commands were executed. Due to the lack of contextual information in *bash_history* files, Ngan proposed the use of a *Window of Command*, Code 3.5, grouping three consecutive commands to simulate the context of execution.

```python
def create_n_command_by_sliding_window(session_cmds, labels, window_size=3):
    cmds_flat = []
    labels_flat = []
    for sess_idx, cmds in enumerate(session_cmds):
        for i in range(len(cmds) + 1):
            start_idx = max(0, i - window_size)
            if start_idx == i:
                continue
            cmds_flat.append(" ".join(cmds[start_idx:i]))
            labels_flat.append(labels[sess_idx])
    return cmds_flat, labels_flat
```

Code 3.5: Ngan's window of commands

3.1.3 Test's Order

As mentioned in Section 3.1.2, the dataset was found to be significantly imbalanced, with the GTFOBins portion representing only 1.601% of the unique commands and 3.04% of the total dataset. This imbalance suggests that a model predicting solely the majority class could achieve an accuracy of approximately 97%, despite failing to correctly classify minority instances. To address this issue, five algorithms—ROS, RUS, SMOTE, Near Miss, and OSS—were tested, as each offers distinct advantages and drawbacks. As discussed in Section 2.1.3, ROS increases the representation of the minority class by duplicating existing data, while RUS reduces the majority class by randomly discarding data points. However, as noted by Weiss (94), these approaches have inherent limitations: RUS may result in the loss of critical information, whereas ROS increases the risk of overfitting by repeatedly using the same data. To mitigate these shortcomings, advanced techniques like SMOTE were employed. According to Alamsyah et al. (96), SMOTE generates synthetic samples for the minority class by interpolating between existing data points, balancing the dataset without relying solely on duplication. Near Miss, in contrast, removes majority class samples based on their proximity to the minority class, preserving essential information in the process. Finally, OSS, as described by Kubat (66), eliminates majority class samples that are distant from the minority class, thereby enhancing the minority class representation.

To automate the data balancing process, two Python functions were developed. These functions create balanced datasets, split the data into training and testing sets, and train the model with the chosen balancing

technique. The function *criar_dataset_balanceado* applies all five tested techniques using *random_state = 32* to ensure reproducibility across methods, as shown in Code 3.6.

```
def criar_dataset_balanceado(X, Y):
    rus = RandomUnderSampler(random_state=32)
    X_rus_res, y_rus_res = rus.fit_resample(X, Y)
    nm = NearMiss(version=1)
    X_nm_res, y_nm_res = nm.fit_resample(X, Y)
    oss = OneSidedSelection(random_state=32)
    X_oss_res, y_oss_res = oss.fit_resample(X, Y)
    ros = RandomOverSampler(random_state=32)
    X_ros_res, y_ros_res = ros.fit_resample(X, Y)
    smote = SMOTE(random_state=32)
    X_smote_res, y_smote_res = smote.fit_resample(X, Y)
    return X_rus_res, y_rus_res, X_nm_res, y_nm_res, X_oss_res, y_oss_res,
        X_ros_res, y_ros_res, X_smote_res, y_smote_res
```

Code 3.6: Creating a balanced dataset

As for the NLP processing used, similar to Ngan's approach, three techniques were employed: Bag of Words (BoW), Term Frequency-Inverse Document Frequency (TF-IDF), and Doc2Vec. All of these techniques were configured to use a feature size of 128. To ensure reproducibility, the code for training each NLP model is provided below, divided into separate listings for clarity in Code 3.7, 3.8 and 3.9.

```
from sklearn.feature_extraction.text import CountVectorizer
vocab_size = 128
vectorizer = CountVectorizer(max_features=vocab_size)
vectorizer.fit(one_cmd_corpus)
X_train_encoded = vectorizer.transform(cmds_flat)
```

Code 3.7: Training Bag of Words (BoW) with 128 vocab size

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.decomposition import TruncatedSVD
VOCAB_SIZE = 128
N_GRAM = 3
vectorizer = CountVectorizer(ngram_range=(1, N_GRAM), max_features=VOCAB_SIZE)
vectorizer.fit(X)
X_train_encoded = vectorizer.transform(X)
tf_transformer = TfidfTransformer().fit(X_train_encoded)
X_train_tfidf = tf_transformer.transform(X_train_encoded)
svd = TruncatedSVD(n_components=50).fit(X_train_tfidf)
X_train_features = svd.transform(X_train_tfidf)
```

Code 3.8: Training TF-IDF with 128 vocab size and SVD

```python
from gensim.models import doc2vec
all_corpus = []
for cmd, lbl in zip(cmds_flat, labels_flat):
    tokens = cmd.split()
    all_corpus.append(doc2vec.TaggedDocument(tokens, str(lbl)))
VECTOR_SIZE = 128
model = doc2vec.Doc2Vec(
    vector_size=VECTOR_SIZE,
    min_count=2,
)
model.build_vocab(all_corpus)
model.train(all_corpus, total_examples=model.corpus_count, epochs=model.epochs)
def extract_features(tagged_corpus, model):
    X = []
    y = []
    for words, tags in tagged_corpus:
        X.append(model.infer_vector(words))
        y.append(float(tags))
    return X, y
X_train_features, y = extract_features(all_corpus, model)
```

Code 3.9: Training Doc2Vec with 128 vector size

Regarding the ML models, Ngan's approach relied solely on KNN and Linear SVC. However, in our reevaluation of Ngan's work, we extended the analysis to include additional ML models. Our findings revealed that several of these models outperformed those previously used. As highlighted in Table 3.2, Random Forest and Neural Networks demonstrated superior performance compared to the other algorithms tested.

NLP Technique	Algorithm	Accuracy	Execution Time	False Negatives
BoW	Decision Tree	93.12%	0.4s	5,270
	Random Forest	93.13%	20.4s	5,271
	Logistic Regression	91.75%	5.4s	4,415
	SVM (SVC)	91.33%	92m 3.5s	6,724
	Neural Network	93.13%	4m 19.6s	5,288
TF-IDF	**Unigram:**			
	Decision Tree	99.49%	8.3s	761
	Random Forest	99.51%	2m 27.1s	766
	Logistic Regression	98.22%	6.4s	2,538
	SVM (SVC)	98.64%	19m 36s	1,573
	Neural Network	99.49%	16m 2.7s	775
	Trigram:			
	Decision Tree	98.87%	7.3s	2,006
	Random Forest	98.88%	1m 23.9s	2,008
	Logistic Regression	98.40%	6.4s	2,255
	SVM (SVC)	98.34%	23m 20.9s	2,145
	Neural Network	98.86%	21m 32.6s	2,023
Doc2Vec	Decision Tree	97.50%	5m 2.5s	3,123
	Random Forest	98.48%	50m 30.8s	831
	Logistic Regression	98.11%	12.6s	920
	SVM (SVC)	98.25%	259m 45.6s	884
	Neural Network	99.23%	197m 37.4s	792

Table 3.2: Performance comparison of algorithms across NLP techniques.

For this reason, we used these five machine learning models to evaluate their performance in classifying malicious LOLBins commands. Ultimately, all combinations of balancing techniques, NLP methods, and ML models, as outlined in Table 3.3, were tested, resulting in a total of 75 experiments.

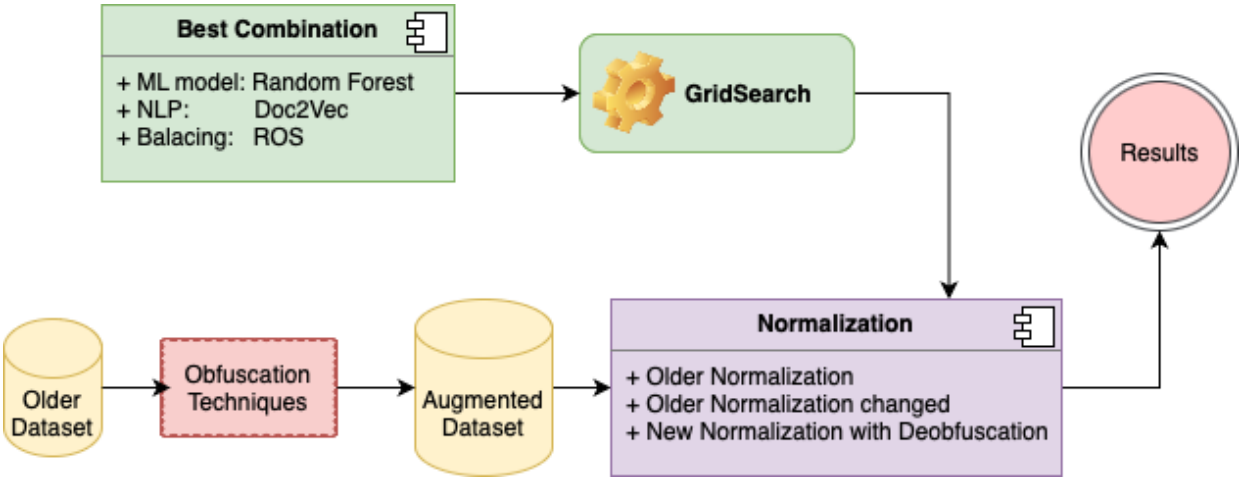
Aspect	Techniques/Models Used
Balancing Techniques	- Random Oversampling (ROS) - Random Undersampling (RUS) - SMOTE - Near Miss - One-Sided Selection (OSS)
NLP Methods	- Bag of Words (BoW) - Term Frequency-Inverse Document Frequency (TF-IDF) - Doc2Vec
ML Models	- Decision Tree - Random Forest - Logistic Regression - Support Vector Machine (SVM) - Neural Network

Table 3.3: Summary of Balancing Techniques, NLP Methods, and ML Models Used

3.2 CLASSIFICATION OF OBFUSCATED LOLBINS IN LINUX

After completion of the processes described in Section 3.1, the model with the best performance, as determined by the MCC score, was selected for fine-tuning. The original dataset used in the initial training was then augmented with obfuscated commands, resulting in a significantly larger dataset. As seen in Figure 3.3 this enhanced dataset was subsequently used to train the fine-tuned model using three different normalization techniques. The first was the same method applied during the initial training, the second introduced slight modifications to handle strings more effectively, and the third employed a novel normalization approach designed to deobfuscate the commands prior to training.

Figure 3.3: Overview of the second test.



Source: Author

This section is structured into three parts: *Fine Tuning*, *Obfuscation Techniques*, *Augmented Dataset* and *Normalization Functions*.

3.2.1 Fine Tuning

All results will be detailed in Section 4, but to address aspects of the training process, some findings from the previous training phase need to be highlighted. It was determined that the combination yielding the best MCC score comprised the Random Forest model, Doc2Vec for feature representation, and Random Oversampling for balancing the dataset.

The Random Forest (RF) model used for binary classification of LOLBins was configured with a straightforward setup: *random_state* set to 0, Gini impurity as the splitting criterion, and the number of trees in the forest (*n_estimators*) set to 100. To optimize these parameters, the *GridSearchCV* function from *sklearn* was employed. This method applied the Cross Validation technique to evaluate multiple parameter combinations and identify the optimal configuration. As shown in Code 3.10, the parameters *criterion*, *min_samples_leaf*, *min_samples_split*, and *n_estimators* were tested. This process systematically evaluates all possible parameter combinations within the specified range and returns the configuration that delivers the best performance.

```
parametros = {
    'criterion' : ['gini', 'entropy'],
    'n_estimators' : [40, 100, 200],
    'min_samples_split' : [2, 5, 10],
    'min_samples_leaf' : [1, 5, 10]
}
grid = GridSearchCV(estimator=RandomForestClassifier(), param_grid=parametros)
grid.fit(X_ros_res, y_ros_res)
```

Code 3.10: Grid Search for Random Forest

3.2.2 Obfuscation Techniques

As discussed in Section 2.2.2, and specifically illustrated in Table 2.6, the syntax used in interpreting shell commands in Linux allows obfuscation techniques similar to those described by Tsai (90) in a PowerShell environment. To handle obfuscated data during text pre-processing, a methodology was employed to standardize it with non-obfuscated strings. This involved creating Python functions to enhance the *normalize* function, ensuring consistency across the dataset.

The following obfuscation techniques were addressed in this study: *Base64 Encoding, Command Substitution, Variable Manipulation, and Encoding Schemes*. These techniques were systematically processed to mitigate their impact and improve the effectiveness over the machine learning models.

3.2.2.1 Base64 Encoding

Base64 can be used to encode and store information, including commands. An example of this is shown in Code 3.11, where the *whoami* command is executed after being encoded in Base64.

```
$ base64 <<< whoami
d2hvYW1pCg==
$ echo "d2hvYW1pCg==" | base64 -d | bash
angellocassio
```

Code 3.11: Executing commands encoded in Base64

To process texts encoded in Base64, a Python function was developed to detect and decode them, as shown in Code 3.12.

```
def decode_base64_if_needed(s):
    base64_pattern = re.compile(r'([A-Za-z0-9+/=]{20,})')
    def decode_base64(match):
        try:
            base64_str = match.group()
            if len(base64_str) % 4 != 0:
                return base64_str
            decoded_bytes = base64.b64decode(base64_str, validate=True)
            return decoded_bytes.decode('utf-8')
```

```
            except (binascii.Error, ValueError, UnicodeDecodeError):
                return match.group()
        decoded_str = base64_pattern.sub(decode_base64, s)
        return decoded_str, decoded_str != s
```

Code 3.12: Detecting and decoding Base64 strings

This function ensures that normal text remains unaffected, as it first verifies Base64 compatibility. According to RFC 4648 (97), a valid Base64-encoded string must consist of an integer number of octets. Specifically: - If the input is a multiple of 24 bits, the final sequence will have a length that is a multiple of 4. - If the input is 8 or 16 bits, the character "=" will be appended once or twice, respectively.

Before attempting decoding with `b64decode`, the function checks if the string's length is divisible by 4, ensuring a valid Base64 size. If decoding fails, an exception is caught, and the function safely returns the original text without alteration.

3.2.2.2 Command Substitution

In Unix systems, it is possible to embed commands within other commands, either to construct an argument or to assign a value to a variable (98). This technique is known as *command substitution*, and it can be implemented using two syntaxes: *$(COMMAND)* or *`COMMAND`*. An example of this is illustrated in Code 3.13, where the *whoami* command is executed as a demonstration.

```
$ echo `whoami`
angellocassio
$ echo $(whoami)
angellocassio
```

Code 3.13: Executing Command Substitution

The original text normalization function previously misprocessed *command substitution* patterns, which could result in the Machine Learning model treating them as entirely different commands. To address this, specialized functions were implemented to correctly normalize such patterns. These functions, shown in Code 3.14, replace command substitution patterns with spaces, ensuring consistency in text processing.

```
def replace_dollar_parentheses(text):
    return re.sub(r'\$$begin:math:text$(.*?)$end:math:text$', r' \1 ', text)

def replace_backticks(text):
    return text.replace('`', ' ')
```

Code 3.14: Replacing Command Substitution Patterns

3.2.2.3 Variable Manipulation

Another method of obfuscating commands involves inserting data into variables and subsequently concatenating them to construct the final command. In Code 3.15, the *whoami* command is split into two variables, which are then concatenated and executed.

```
$ Universidade="wh"
$ deBrasilia="oami"
$ $Universidade$deBrasilia
angellocassio
```

Code 3.15: Executing Variable Manipulation

In some cases, the variable definitions may not be present in the same text segment as the concatenated command, even when employing Ngan's Window of Command technique (13). To address this limitation, variable concatenations are replaced with the placeholder string *'CONCATENACAO'*, ensuring consistent normalization. The implementation of this replacement is shown in Code 3.16.

```
def replace_env_concatenation(text):
    return re.sub(r'(\$\w+(\s*\$\w+)+)', 'CONCATENACAO', text)
```

Code 3.16: Replacing Variable Concatenations

3.2.2.4 Encoding Schemes

The manual page for the *echo* command (99) highlights that the argument *-e* enables the interpretation of backslash escapes, which include representations for Unicode encoding, as well as octal and hexadecimal digits. Unicode is a universal character encoding standard, while octal and hexadecimal are numerical systems used to represent values (100, 97, 101).

By combining the *echo* command with *bash* through a pipe, it is possible to execute commands encoded in these schemes, as demonstrated in Code 3.17.

```
[$] echo -e "\x77\x68\x6f\x61\x6d\x69" | bash
> angellocassio
```

Code 3.17: Executing Encoded Commands

To process encoded strings effectively, the function in Code 3.18 was developed to identify and decode these encoding schemes using regular expressions. The *echo* command interprets octal patterns as *\0nn*, hexadecimal patterns as *\xHH*, and Unicode patterns as *\uHHHH*. The *decode_escapes* function decodes each of these patterns separately, transforming them into a format that can be further processed by the NLP algorithm.

```
def hex_to_char(match):
```

```
    return bytes.fromhex(match.group(1)).decode('utf-8')

def unicode_to_char(match):
    return chr(int(match.group(1), 16))

def octal_to_char(match):
    try:
        return chr(int(match.group(1), 8))
    except ValueError:
        return match.group(0)

def decode_escapes(text):
    text = re.sub(r'\\x([0-9A-Fa-f]{2})', hex_to_char, text)
    text = re.sub(r'\\u([0-9A-Fa-f]{4})', unicode_to_char, text)
    text = re.sub(r'\\([0-3]?[0-7]{2,3})', octal_to_char, text)
    return text
```

Code 3.18: Decoding Encoded Strings

3.2.2.5 Escape Characters and Alias

Two additional techniques that can obscure and obfuscate commands in Linux are the use of *Escape Characters* and *Alias Commands*. According to the Bash manual [3], an escape character is defined as a non-quoted backslash, which preserves the literal value of the next character, except for newlines. On the other hand, the *alias* command [4] is used to create a new name or shortcut for an existing command. Leveraging these techniques, the *whoami* command can be executed as demonstrated in Code 3.19.

```
# Escape Characters
$ w\ho\am\i
angellocassio

# Alias
$ alias myname='whoami'
$ myname
angellocassio
```

Code 3.19: Executing Obfuscated Commands - Escape and Alias

However, these techniques were not incorporated into the new normalization function. Their exclusion is primarily due to the complexity involved in adapting the entire dataset to account for them. As such, they were not used to augment the dataset.

[3] <https://www.gnu.org/savannah-checkouts/gnu/bash/manual/html_node/Escape-Character.html>
[4] <https://www.man7.org/linux/man-pages/man1/alias.1p.html>

3.2.3 Augmented Dataset

The original dataset did not include encoded commands necessary to perform the tests, making it essential to construct one. For this purpose, functions were created to transform all entries into encoded commands based on the techniques outlined in Section 3.2.2, except for *Escape Characters*, *Alias*, and *Variable Manipulation* (the latter only had a normalization function created). The implemented functions, shown in Code 3.20, Code 3.21, and Code 3.22, perform transformations for different encoding and substitution techniques.

```
def convert_command_to_hex(s):
    return ''.join(f'\\x{ord(c):02x}' for c in s)
def convert_command_to_octal(s):
    return ''.join(f'\\0{ord(c):03o}' for c in s)
def convert_command_to_unicode(s):
    return ''.join(f'\\u{ord(c):04x}' for c in s)
```

Code 3.20: Transforming Dataset - Encoding Commands

```
def transform_strings_to_base64_bash(strings):
    new_strings = []
    for s in strings:
        encoded_bytes = base64.b64encode(s.encode('utf-8'))
        encoded_str = encoded_bytes.decode('utf-8')
        new_str = f"echo {encoded_str} | base64 -d | bash"
        new_strings.append(new_str)
    return new_strings
```

Code 3.21: Transforming Dataset - Base64 Encoding

```
def generate_alternatives(strings):
    new_strings = []
    for s in strings:
        new_str_dollar = f"$({s})"
        new_str_backticks = f"`{s}`"
        new_strings.append(new_str_dollar)
        new_strings.append(new_str_backticks)
    return new_strings
```

Code 3.22: Transforming Dataset - Command Substitution

3.2.4 Normalization Functions

The primary goal of this phase is to conduct three separate trainings using different normalization techniques, as prior studies have demonstrated that normalization significantly impacts model performance. The first normalization approach employs the same function used in the initial part of this work to ensure a fair comparison with the newer techniques.

The second normalization method differs from the first in handling quoted strings. In the original approach, all text enclosed in quotation marks was replaced with the placeholder *_STRING_*. However, as shown in Code 3.23, the new function removes the quotation marks while retaining the text inside them. This allows the ML model to leverage the quoted content as useful information during training.

```
def replace_quoted_string(text):
    return re.sub(r'"(.*?)"|\'(.*?)\'', lambda m: m.group(1) if m.group(1) is not
        None else m.group(2), text)
```

Code 3.23: New Function for Handling Quoted Strings

The final test involves deobfuscating all commands before sending them to the NLP model for feature extraction. This ensures that the obfuscation does not hinder the ability of the model to analyze and classify the commands effectively. For Command Substitution, all dollar-parentheses and backtick notations are removed using regular expressions, as shown in Code 3.24.

```
def replace_dollar_parentheses(text):
    return re.sub(r'\$\((.*?)\)', r' \1 ', text)
def replace_backticks(text):
    return text.replace('`', ' ')
```

Code 3.24: Deobfuscation Function - Command Substitution

As shown in Code 3.25, for Base64 obfuscation, the function first identifies potential Base64 strings using a regular expression pattern. It then verifies if the string adheres to the Base64 length requirements specified in RFC 4648. If the string passes the validation, it is decoded; otherwise, it remains unchanged.

```
def decode_base64_if_needed(s):
    base64_pattern = re.compile(r'([A-Za-z0-9+/=]{20,}={0,2})')

    def decode_base64(match):
        try:
            base64_str = match.group()
            if len(base64_str) % 4 != 0:
                base64_str += '=' * (4 - len(base64_str) % 4)
            decoded_bytes = base64.b64decode(base64_str, validate=True)
            return decoded_bytes.decode('utf-8')
        except (binascii.Error, ValueError, UnicodeDecodeError):
            return match.group()  # Return original text if decoding fails

    decoded_str = base64_pattern.sub(decode_base64, s)
    return decoded_str, decoded_str != s
```

Code 3.25: Deobfuscation Function - Base64

Finally, in Code 3.26 provides a function to decode the hexadecimal, octal and unicode strings if the text matches a certain regex. The function applies regular expressions to identify specific patterns

corresponding to these encodings. If a match is found, the respective decoding logic is executed, converting the encoded values back into their readable string representation.

```
def hex_to_char(match):
    return bytes.fromhex(match.group(1)).decode('utf-8')
def unicode_to_char(match):
    return chr(int(match.group(1), 16))
def octal_to_char(match):
    try:
        return chr(int(match.group(1), 8))
    except ValueError:
        return match.group(0)
def decode_escapes(text):
    text = re.sub(r'\\x([0-9A-Fa-f]{2})', hex_to_char, text)
    text = re.sub(r'\\u([0-9A-Fa-f]{4})', unicode_to_char, text)
    text = re.sub(r'\\([0-3]?[0-7]{2,3})', octal_to_char, text)
    return text
```

Code 3.26: Deobfuscate Function - Encoded Commands

All these new functions, along with the one cited in Code 3.16, are executed prior to the original normalization function. This ensures that all commands are fully deobfuscated before undergoing proper normalization.

4 DATA ANALYSIS AND RESULTS

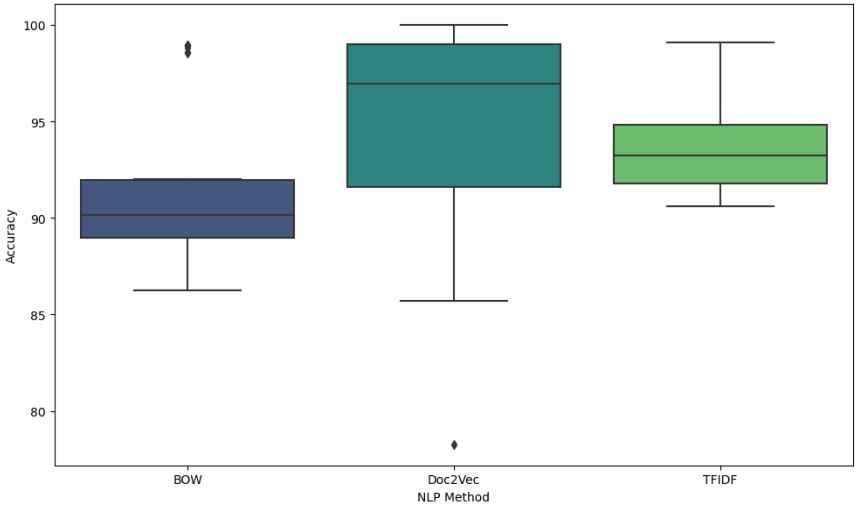
This chapter presents the results of the experiments conducted in this study. It is divided into two main sections: the classification of plain LOLBins and the classification of obfuscated LOLBins.

4.1 RESULTS OF THE PLAIN LOLBINS CLASSIFICATION

For each scenario, five training sessions were conducted, corresponding to five balancing algorithms across six supervised learning models. The performance of the NLP techniques, evaluated by various metrics, is presented in Box Plot graphs for better visualization and comparison.

Figure 4.1 illustrates the accuracy distribution for each NLP method. Doc2Vec exhibited the broadest range of accuracy values, indicating significant variability depending on the choice of ML model and balancing algorithm. In contrast, TF-IDF and BoW demonstrated more consistent accuracy, with BoW showing a few prominent outliers. These results suggest that while Doc2Vec has the potential to achieve high accuracy, its performance is more sensitive to the choice of complementary techniques. In contrast, BoW and TF-IDF offer more stable and predictable results.

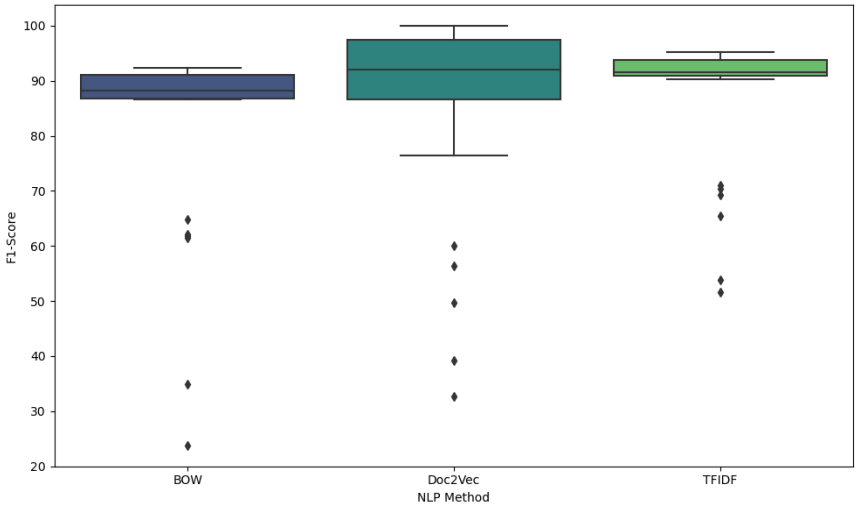
Figure 4.1: Box Plot of accuracy across NLP methods.



Source: Author

Figure 4.2 presents the F1-Score distribution for each NLP method. Similar to the accuracy metric, Doc2Vec displayed the widest range of F1-Score values, emphasizing its variability across different configurations. Both BoW and TF-IDF showed greater stability, with BoW exhibiting slightly more outliers than TF-IDF.

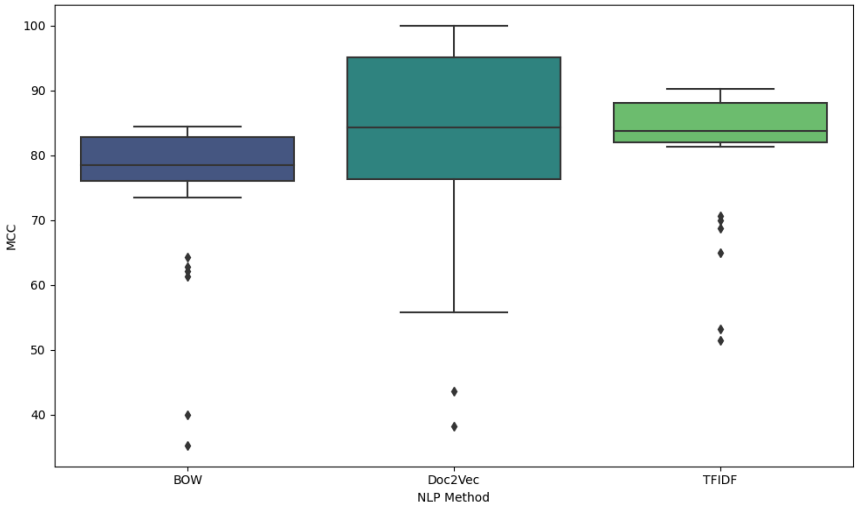
Figure 4.2: Box Plot of F1-Score across NLP methods.



Source: Author

Finally, the MCC score distribution, the primary evaluation metric for this work, is shown in Figure 4.3. Once again, Doc2Vec exhibited the widest performance range, reflecting its sensitivity to the ML model and balancing algorithm.

Figure 4.3: Box Plot of MCC scores across NLP methods.



Source: Author

However, this variability does not imply that Doc2Vec is the weakest performer. As shown in Table 4.1, while Doc2Vec demonstrated the most unstable performance across all metrics, it also achieved the best results when properly optimized. Conversely, TF-IDF was the most stable option, offering consistent performance across scenarios. BoW, despite its stability, exhibited a high number of outliers, suggesting occasional significant deviations.

Table 4.1: Summary of Metrics by NLP Method

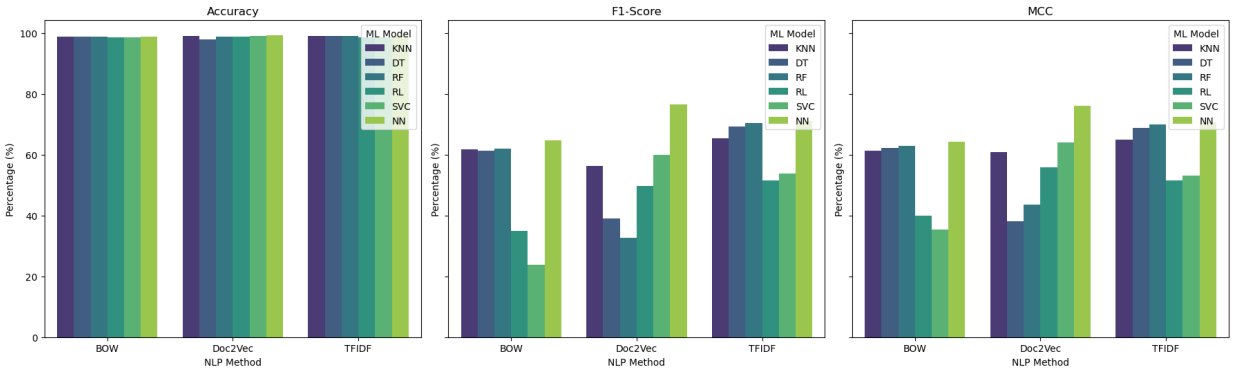
NLP Method	Accuracy (Mean ± Std)	F1-Score (Mean ± Std)	MCC (Mean ± Std)
BoW	90.12 ± 2.35	88.45 ± 5.22	80.34 ± 8.14
Doc2Vec	92.78 ± 5.67	91.34 ± 7.45	85.67 ± 10.23
TF-IDF	89.56 ± 3.12	87.89 ± 4.55	82.45 ± 6.78

The One-Sided Selection (OSS) tests consistently achieved high accuracy across various scenarios, as shown in Figure 4.4. However, it is worth noting that this method's F1-Score and MCC values remained significantly lower.

A low F1-Score indicates that, despite the model's ability to achieve high accuracy by correctly predicting the majority class, it struggles to identify the minority class effectively. Similarly, the low MCC values reinforce this limitation, suggesting that the model's predictions are imbalanced and skewed towards the majority class. This limitation is particularly critical in intrusion detection scenarios, where malicious inputs typically form a minority class, deviating from standard patterns. Accurate identification of such anomalies is essential for ensuring robust security measures.

The observed shortcomings highlight that while OSS may be effective in improving class distribution, its performance in handling imbalanced datasets remains suboptimal for applications that demand precise detection of minority class instances.

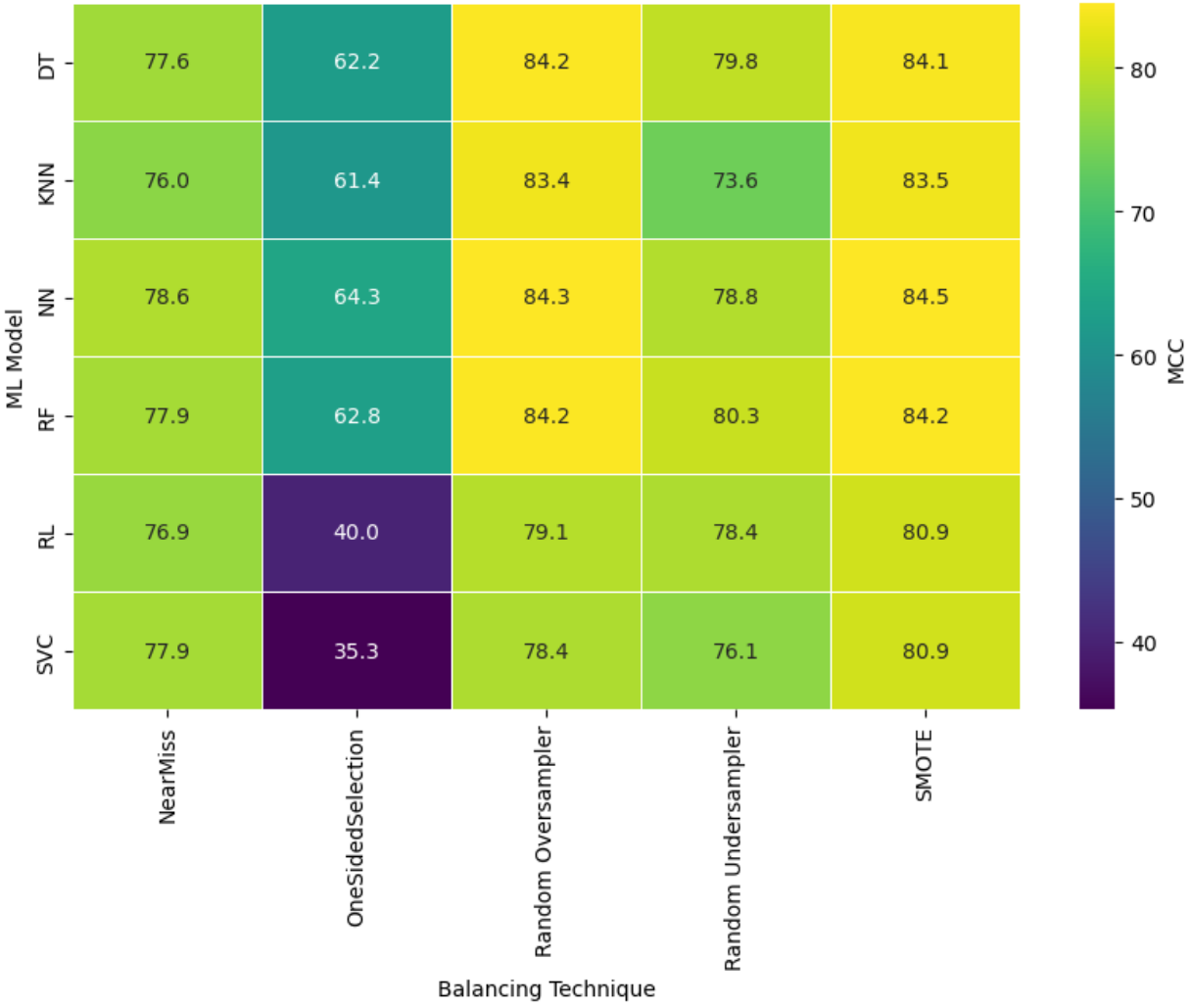
Figure 4.4: Performance metrics for One-Sided Selection (OSS).



Source: Author

In the experiments conducted, algorithms designed to increase the representation of the minority class, particularly ROS and SMOTE, demonstrated superior performance compared to those aiming to reduce the majority class. The heatmap analysis, illustrated in Figure 4.5, showcases the performance of each combination used with the BoW NLP method. The column corresponding to the OSS technique stands out for its poor performance, with the highest MCC reaching only 62.8. Conversely, ROS and SMOTE achieved the best results in this scenario. Although underperformed, the overall MCC range for BoW was relatively stable, varying between 76.0 and 84.5, demonstrating its consistency across different balancing techniques despite OSS.

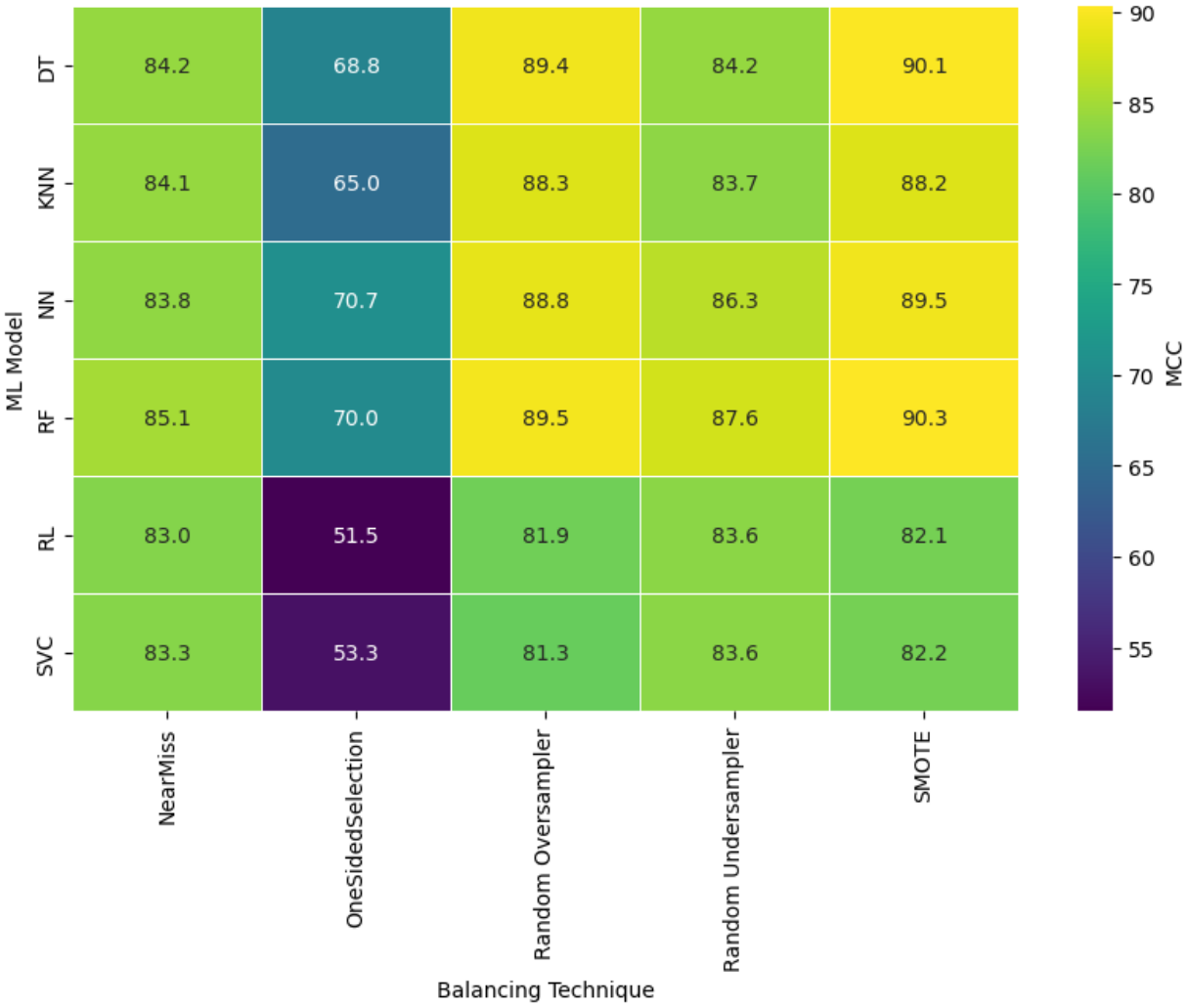
Figure 4.5: Heatmap with the BoW performance.



Source: Author

Figure 4.6 presents the performance of the TF-IDF NLP method. In this case, scores slightly improved compared to BoW, achieving a maximum MCC of 90.3 with SMOTE and the Random Forest (RF) model. TF-IDF consistently outperformed BoW across most combinations, indicating its effectiveness in text vectorization for this task. The stability observed in BoW is also apparent here, but TF-IDF managed to produce higher peak scores in several scenarios.

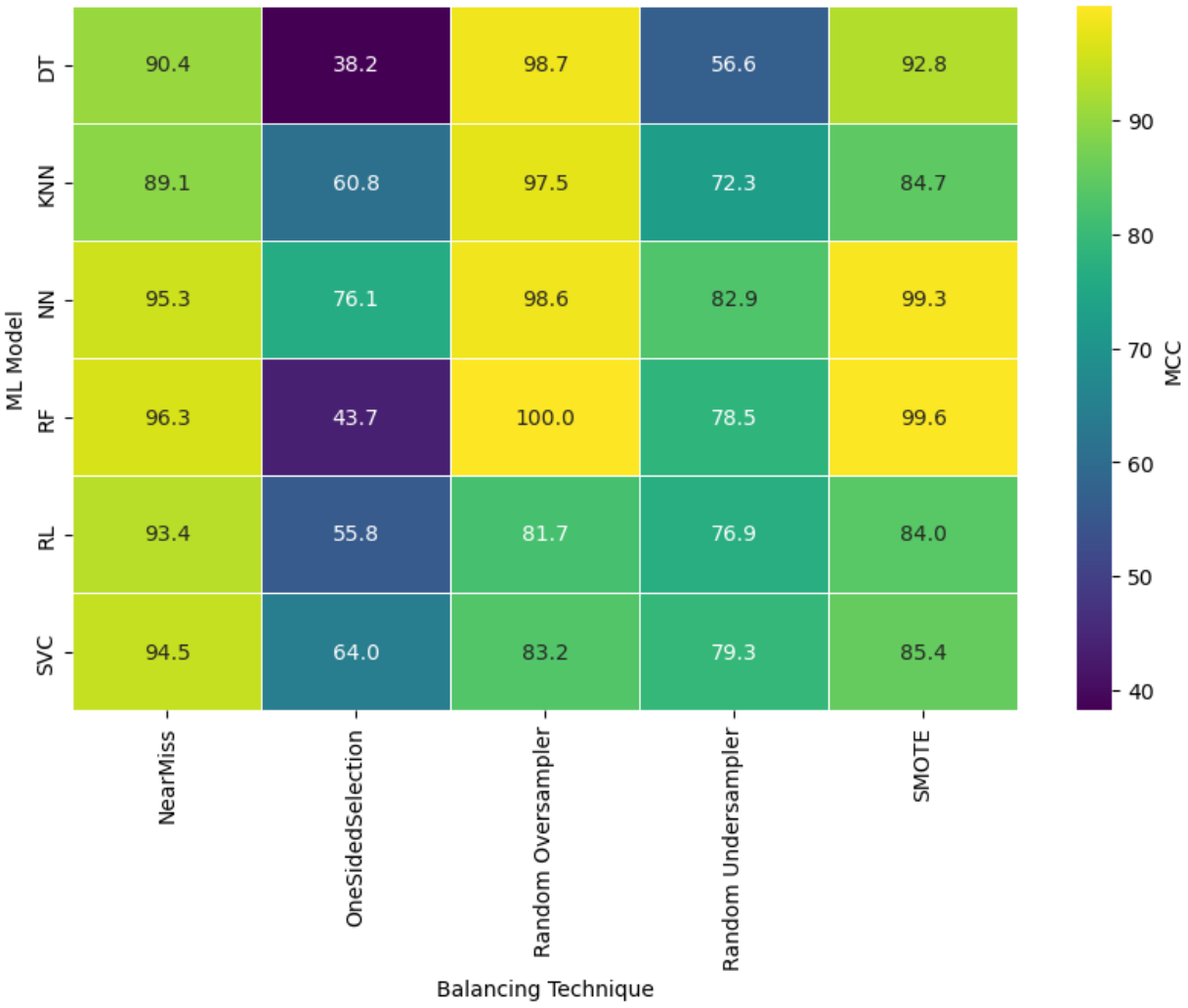
Figure 4.6: Heatmap with the TF-IDF performance.



Source: Author

Finally, Doc2Vec, as illustrated in Figure 4.7, achieved the best performance across all NLP methods, with a maximum MCC score of 99.998, rounded to 100 in the graphic. This exceptional result was obtained using the RF model combined with ROS.

Figure 4.7: Heatmap with the Doc2Vec performance.



Source: Author

Doc2Vec emerged as the most effective NLP method for this classification task, especially when combined with robust balancing techniques such as ROS or SMOTE. TF-IDF demonstrated solid and consistent performance, making it a strong alternative in scenarios where stability and reliability are prioritized. BoW, while stable, showed lower peak performance compared to Doc2Vec and TF-IDF, showing the importance of careful selection of NLP methods in classification tasks.

Table 4.2 shows that the five best evaluations were achieved using Doc2Vec. Regarding machine learning algorithms, Decision Trees (DT), Random Forest (RF), and Neural Networks (NN) consistently demonstrated superior performance, not only with Doc2Vec but also when paired with other NLP techniques.

Table 4.2: Top 5 Performances Achieved in the Evaluation

Technique	Balancing Technique	Model	Metrics (Accuracy | F1-Score | MCC)
Doc2Vec	Random Oversampler	RF	99.9992 | 99.9992 | 99.9984
Doc2Vec	SMOTE	RF	99.8091 | 99.8083 | 99.6182
Doc2Vec	SMOTE	NN	99.6475 | 99.6470 | 99.2962
Doc2Vec	Random Oversampler	DT	99.3544 | 99.3561 | 98.7171
Doc2Vec	Random Oversampler	NN	99.3116 | 99.3118 | 98.6264

Table 4.3 presents the execution time of each training performed in this phase. Algorithms like DT and LR showed significantly lower execution times across all representations, indicating greater computational efficiency. On the other hand, algorithms like SVM and NN require considerably longer execution times. When comparing the NLP representations, the BoW approach resulted in the shortest times for most algorithms, except for the Neural Network, which had a high execution time even in this configuration. The TF-IDF representation, although more expressive regarding classification performance, required more time in algorithms such as SVM. Doc2Vec was a significant computational expense, especially for the SVM and NN algorithms.

Algorithm	BoW	TF-IDF	Doc2Vec
KNN	16min 13s	42.9s	1min 21s
DT	1.79s	14.6s	1min 44s
RF	42.4s	3min 38s	10min 49s
LR	6.05s	5.47s	21.1s
SVM	46min 56s	1h 23min 39s	4h 8min 9s
NN	19min 7s	9min 48s	14min 49s

Table 4.3: Execution time of training

4.2 RESULTS OF THE OBFUSCATED LOLBINS CLASSIFICATION

In Section 4.1, the top five results were achieved using Doc2Vec as the NLP method. Among the balancing algorithms, those that increased the representation of the minority class, such as Random Oversampling and SMOTE, demonstrated superior performance compared to others. As shown in Table 4.2, the highest MCC achieved was 99.992%. Specifically, the best combination utilized Random Oversampling, Doc2Vec, and Random Forest. Based on this, a fine-tuning process was conducted using GridSearch, resulting in the optimal parameters for the Random Forest model, as detailed in Code 4.1.

```
1 # Optimal Parameters
2 {'criterion': 'entropy', 'min_samples_leaf': 5, 'min_samples_split': 5, '
      n_estimators': 200}
```

Code 4.1: Optimal parameters for Random Forest with a vector size of 128

To comprehensively evaluate the impact of the new preprocessing approach, the original dataset was

enhanced with encoded LOLBin commands. Each LOLBin command was systematically transformed to incorporate all the previously discussed encoding techniques, except Variable Manipulation. The Table 4.4 gives an example of the new entries for the command *whoami* in the new dataset.

Table 4.4: Transformations Applied to LOLBin Commands

Command	**Encoding**
`whoami`	Original
`echo -e "\x77\x68\x6f\x61\x6d\x69" | bash`	Hexadecimal
`echo -e "\0167\0150\0157\0141\0155\0151" | bash`	Octal
`echo -e "\u0077\u0068\u006f\u0061\u006d\u0069" | bash`	Unicode
`echo "d2hvYW1p" | base64 -d | bash`	Base64
`$(whoami)`	Command Substitution

Following the transformations and the application of Ngan's Window of Command, the dataset expanded significantly. Initially, the Doc2Vec processing generated 128 features, resulting in a highly demanding dataset. Although the machine used for testing was equipped with 96 GB of RAM, it was not possible to process the dataset with all 128 features in the second training. Consequently, the number of features was reduced to 64. Even with this reduction, the second training took approximately 35 hours, as shown in Table 4.5.

Script	**Execution Time**	**N-commands**
Training 01	9h 15min 7.43s	12.357.660
Training 02	34h 50min 18.65s	37.521.664
Training 03	12h 21min 34.36s	18.251.164

Table 4.5: Execution time and dataset size (N-commands) for each training script

As discussed in Section 3.2, the new dataset was preprocessed in three distinct ways. First, it was processed using the original *normalize* function, preserving the same approach used in previous tests. Second, the *normalize* function was modified so that the *replace_quoted_string* routine would only remove single or double quotation marks, allowing the content within them to be analyzed by the NLP algorithm. Finally, the dataset was processed using the newly developed *re_normalize* function, which incorporates all deobfuscation techniques.

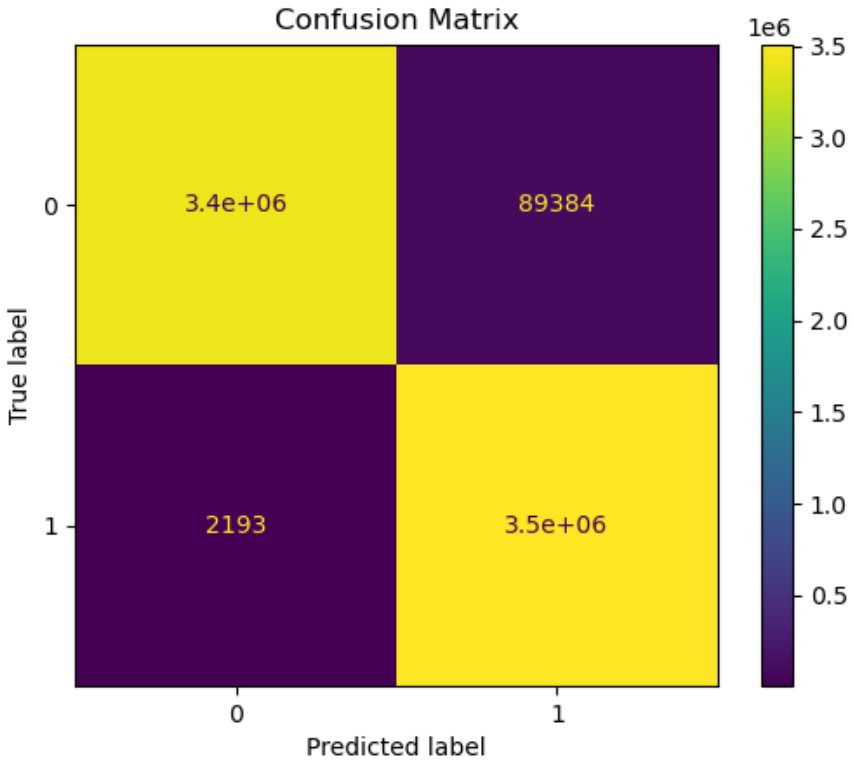
As shown in Table 4.5, these three preprocessing strategies resulted in datasets of different sizes. For this reason, evaluation metrics alone are insufficient to assess their performance. Therefore, confusion matrices were also used for a more comprehensive analysis.

However, this limitation was only identified during the second training. Consequently, the first training was executed with a feature size of 128. Contrary to initial expectations, the model demonstrated strong performance, achieving an accuracy of approximately 98.71% and a Matthews Correlation Coefficient (MCC) of 97.41%. The confusion matrix presented in Figure 4.8 shows that the model correctly classified most benign and malicious commands. Nevertheless, it produced 2,193 false negatives. Although this number represents a small fraction of the total dataset, it is essential to emphasize that, in a real-world detection scenario, these false negatives correspond to undetected malicious commands. This highlights a

critical risk, as it would allow 2,193 attacks to bypass the detection system.

The confusion matrix also shows a relatively low number of false positives (89,384), indicating that the model effectively distinguishes benign commands from malicious ones. However, further fine-tuning or additional preprocessing strategies may be required to reduce the false negative rate, as even a small number of undetected attacks can have significant security implications.

Figure 4.8: Confusion Matrix for Training 01.



Source: Author

During the second training, the approach to handling text within quotation marks significantly impacted the dataset size. The normalized dataset generated in this stage contained 37,521,664 entries, approximately three times larger than the dataset used in the first training, which had 12,357,660 entries. As a result of this substantial increase, the machine used for testing was unable to process the dataset efficiently.

As a result, it was necessary to revisit the fine-tuning process. The vector size for Doc2Vec was reduced from 128 to 64 to accommodate the hardware limitations. Despite this adjustment, the optimization process yielded slightly different results. The new optimal parameters for the Random Forest model, shown in Code 4.2, varied only in the value for *n_estimators*, while maintaining the same *criterion*, *min_samples_split* and *min_samples_leaf*.

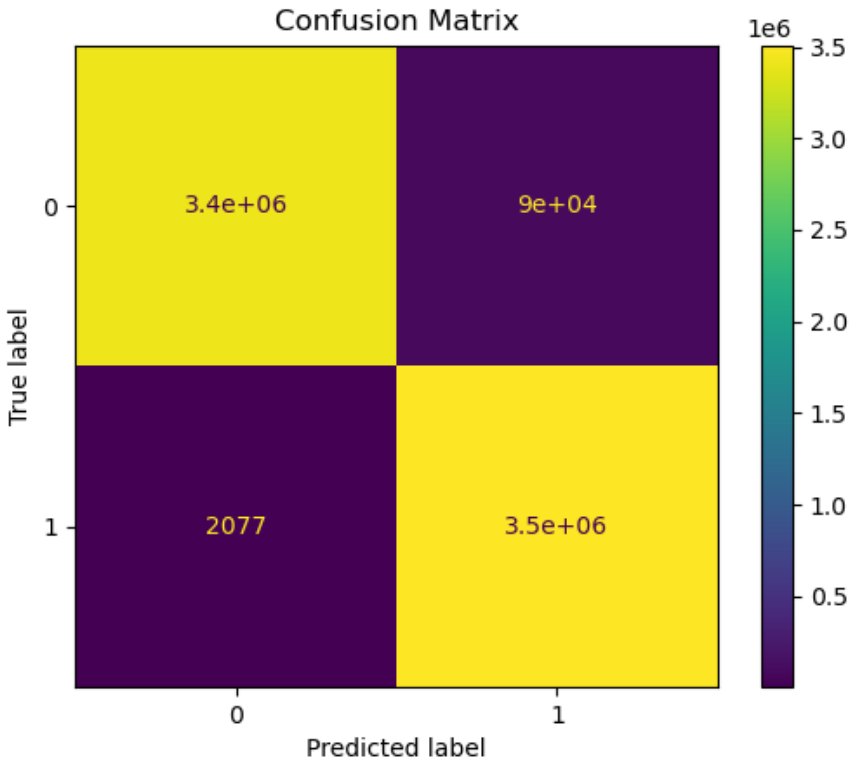
```
1 # New Optimal Parameters
2 {'criterion': 'entropy', 'min_samples_leaf': 5, 'min_samples_split': 5, '
```

```
        n_estimators': 100}
```

Code 4.2: Optimal parameters for Random Forest with a vector size of 64

The first training was re-executed with the adjusted parameters and a reduced vector size of 64. The results showed minimal differences compared to the previous test, with a vector size of 128. As depicted in the Confusion Matrix in Figure 4.9, the MCC score achieved was 97.3969, slightly lower than the 97.4196 obtained with the larger vector size. This negligible variation suggests that the reduction in vector size did not significantly impact the model's performance.

Figure 4.9: Confusion Matrix for Training 01 with a vector size of 64.

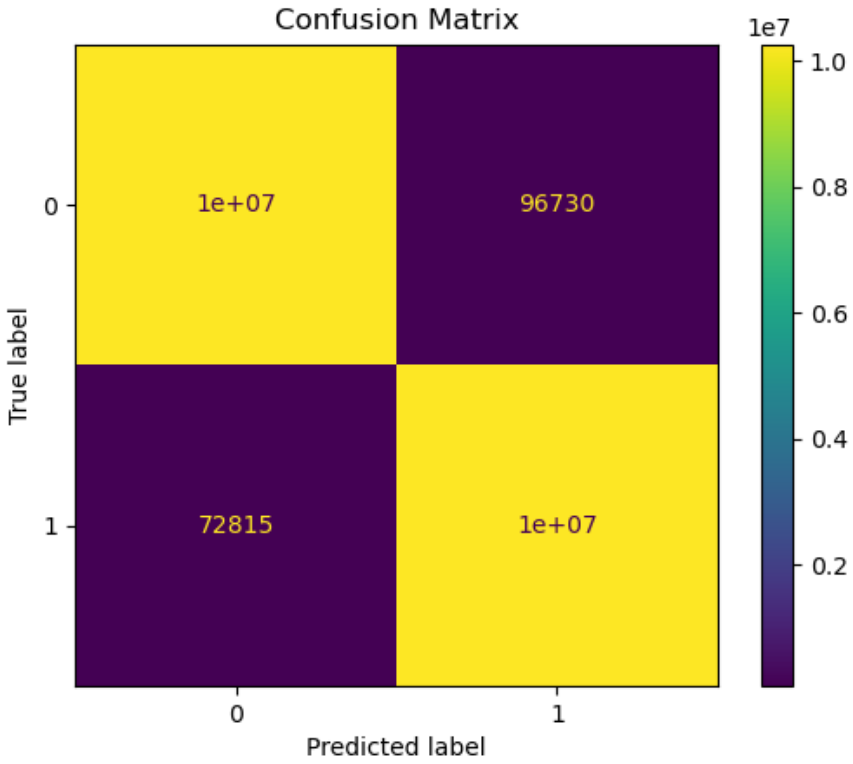


Source: Author

In the second scenario, as previously mentioned, the differences in preprocessing the text strings resulted in a dataset that was three times larger than the original. However, the model successfully completed the routine with a reduced vector size of 64. The MCC score saw a modest increase of approximately 1%, reaching 98.3591%, while the F1-Score improved to 99.1594%.

Despite the improved scores, the Confusion Matrix in Figure 4.10 reveals an increase in the number of False Negatives, totaling 72,815 mismatches. This indicates that while the overall performance metrics improved, the model allowed more malicious commands to go undetected.

Figure 4.10: Confusion Matrix for Training 02 with a vector size of 64.

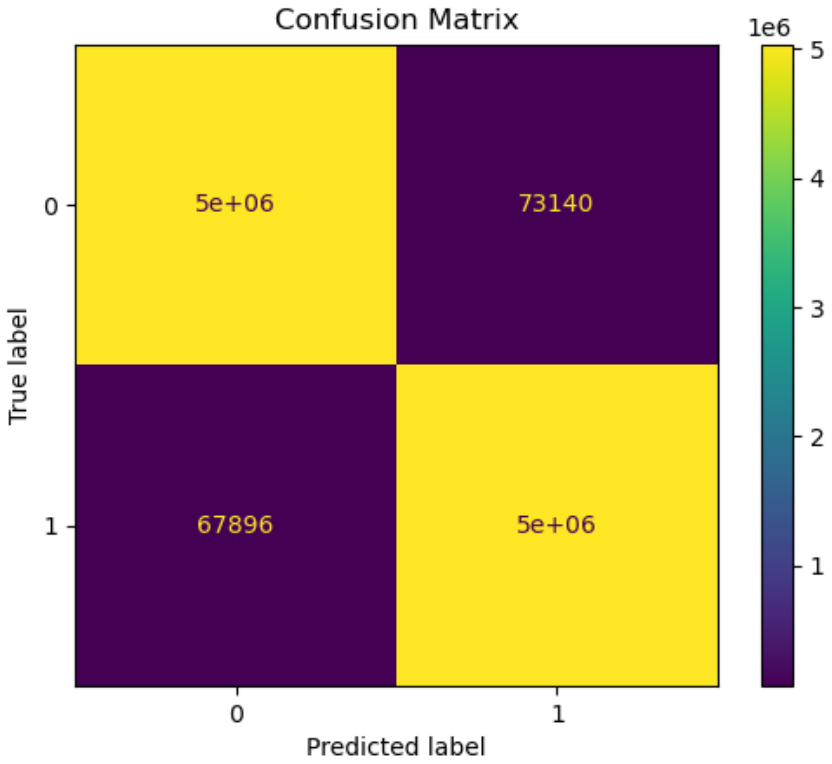


Source: Author

Finally, in the third and final training, the dataset size was slightly more prominent than in the first training but smaller than in the second. Despite initial expectations that the deobfuscation techniques would enhance the detection rate, this test resulted in the weakest performance among the three experiments, achieving an MCC of 97.2186%.

As depicted in the Confusion Matrix in Figure 4.11, the number of False Negatives remained almost as high as in the second training, further emphasizing the limitations of this approach. While the deobfuscation process was intended to improve the model's ability to detect malicious commands, the results suggest that it may introduce complexity that adversely impacts the model's performance.

Figure 4.11: Confusion Matrix for Training 03 with a vector size of 64.



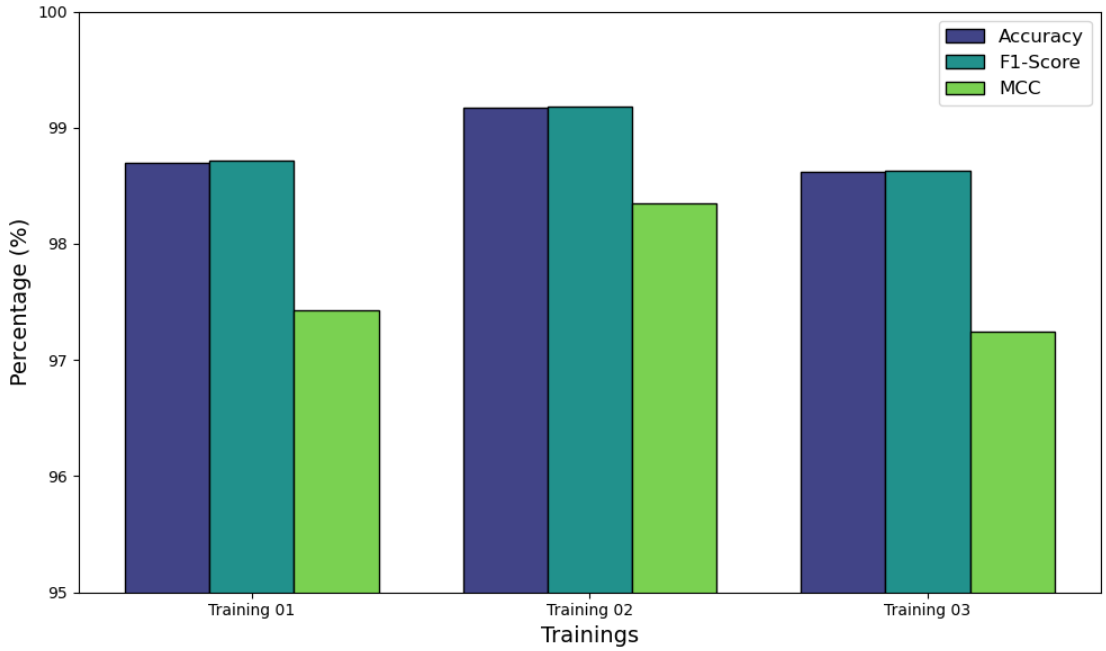
Source: Author

When comparing the MCC scores across the three training sessions, it becomes evident that the difference between the highest and lowest scores was less than 1%, as shown in Table 4.6 and illustrated in Figure 4.12.

Table 4.6: Performance Metrics for All Trainings

Metric	**Training 01**	**Training 02**	**Training 03**
Accuracy	98.69097095270440%	99.18037875474455%	98.62713069482673%
F1-Score	98.70729753929967%	99.18116701538018%	98.62815753036448%
MCC	97.41256167550615%	98.36090871364883%	97.25435446805292%

Figure 4.12: Performance Metrics for Final Trainings.



Source: Author

This minor variation suggests that introducing obfuscated alternatives for each command in the dataset may have introduced a bias in the model. This bias could limit the model's ability to accurately differentiate between the tested scenarios, thereby affecting its capacity to identify the optimal approach for improving performance.

5 FINAL CONSIDERATIONS

5.1 CONCLUSION

Building on the innovative foundation laid by Ngan's work, this research extended the study of the classification of Malicious Linux Commands by exploring additional machine learning algorithms and incorporating the Doc2Vec NLP method. This study demonstrated that combining alternative NLP techniques and machine learning models can offer better solutions tailored to the problem being addressed. Additionally, improvements were made to the normalization process to prevent the loss of critical information during preprocessing.

Using an enhanced version of Ngan's methodology, this research successfully classified LOLBins in their plain form, addressing the significant challenge of an imbalanced dataset. The findings emphasize the importance of combining NLP techniques, effective balancing algorithms, and machine learning models to achieve accurate classification results. By testing various combinations, the study identified that Doc2Vec yielded the most effective results when paired with Random Forest and oversampling techniques like ROS and SMOTE.

The evaluation highlighted that techniques aimed at reducing the majority class often resulted in significant information loss, negatively impacting overall performance. On the other hand, oversampling techniques such as ROS and SMOTE demonstrated their effectiveness in balancing the dataset without compromising classification accuracy. While Doc2Vec achieved the highest metrics under optimal conditions, TF-IDF proved to be a more consistent alternative across different scenarios. Meanwhile, though stable, traditional approaches like BoW fell short in delivering competitive performance, underscoring the importance of selecting NLP methods that align with specific use cases.

After identifying the best-performing combination of methods, the machine learning model was fine-tuned using the Grid Search method to optimize its parameters. The study then progressed to a more challenging task: classifying obfuscated LOLBins. The dataset was expanded with obfuscated versions of the original entries, incorporating multiple obfuscation techniques to achieve this. Three normalization approaches were tested, including one that deobfuscated the commands before NLP processing. While this approach added complexity, it did not significantly enhance performance, revealing the challenges of addressing obfuscation in real-world datasets.

The study's exploration of preprocessing techniques and obfuscation highlighted the trade-offs in balancing sophistication with dataset integrity. While adding deobfuscation functions enriched the dataset, it introduced diminishing returns in classification performance. This suggests that overly complex preprocessing may inadvertently introduce biases or inconsistencies, emphasizing the need for careful dataset preparation. Future research could focus on creating balanced datasets that include a diverse range of obfuscated and plain commands to evaluate detection models better.

This work also revealed the computational limitations of current methods. The significant time and resources required for fine-tuning and training on large datasets underscore the need for more scalable

solutions in cyber intelligence. Future studies could explore optimization techniques, distributed architectures, or hybrid machine learning approaches to handle large-scale data without sacrificing performance.

In conclusion, this research contributes valuable insights into the detection of LOLBins, advancing the understanding of both plain and obfuscated command classification. Addressing these challenges lays the groundwork for further advancements in normalization techniques, robust machine learning models, and scalable architectures. Ultimately, this study not only highlights the need for adaptability and continuous innovation but also inspires us to strengthen cybersecurity defenses against increasingly sophisticated threats.

5.2 FUTURE WORK

To further improve this work and effectively mitigate the bias introduced by dataset transformations, the proposed approach must be evaluated using a more diverse dataset, including both legitimate and malicious LOLBins, in obfuscated and unobfuscated forms.

Another important direction is to enhance the model's performance to accommodate a feature size of 128, as initially intended. This also involves comparing performance across different hardware setups and system architectures since the current research was conducted on an ARM-based architecture in a macOS environment.

Finally, the deobfuscation technique implemented in this work addresses only a single layer of obfuscation. However, an adversary may apply multiple layers of obfuscation to a malicious command. As it currently stands, the function cannot recursively handle such cases. Therefore, extending the deobfuscation process to support multi-layered obfuscation is a key area for future exploration.

BIBLIOGRAPHIC REFERENCES

1 ABUSHQEIR, A. *Common Pattern Generation for the Detection of LOLBin Attacks*. Thesis (Master) — San Jose State University, Fall 2023. Available at: <https://scholarworks.sjsu.edu/etd_theses/5422>.

2 PINNA, E.; CARDACI, A. *GTFOBins*. 2023. Accessed on: DEC. 5, 2023. Available at: <https://gtfobins.github.io>.

3 NING, R.; BU, W.; YANG, J.; DUAN, S. A survey of detection methods research on living-off-the-land techniques. In: *2023 IEEE International Conference on Sensors, Electronics and Computer Engineering (ICSECE)*. [S.l.: s.n.], 2023. p. 159–164.

4 DING, K.; ZHANG, S.; YU, F.; LIU, G. Lolwtc: A deep learning approach for detecting living off the land attacks. In: *2023 IEEE 9th International Conference on Cloud Computing and Intelligent Systems (CCIS)*. [S.l.: s.n.], 2023. p. 176–181. ISSN 2376-595X.

5 PIRANHA, R. *LIVING OFF THE LAND (LOTL) ATTACKS AND DEFENDING AGAINST APT'S*. 2023. Accessed on: DEC. 5, 2023. Available at: <https://redpiranha.net/news/living-off-the-land-and-apts>.

6 BARR-SMITH, F.; UGARTE-PEDRERO, X.; GRAZIANO, M.; SPOLAOR, R.; MARTINOVIC, I. Survivalism: Systematic analysis of windows malware living-off-the-land. In: *2021 IEEE Symposium on Security and Privacy (SP)*. [S.l.: s.n.], 2021. p. 1557–1574. ISSN 2375-1207.

7 MANDIANT. *Barracuda ESG Zero-Day Vulnerability (CVE-2023-2868) Exploited Globally by Aggressive and Skilled Actor, Suspected Links to China*. Mandiant, 2023. Accessed on: JAN. 24, 2024. Available at: <https://www.mandiant.com/resources/blog/barracuda-esg-exploited-globally>.

8 National Institute of Standards and Technology (NIST). *CVE-2023-2868: Barracuda Networks ESG Appliance Improper Input Validation Vulnerability*. 2023. <https://nvd.nist.gov/vuln/detail/CVE-2023-2868>. Accessed on: JAN. 24, 2024.

9 Cybersecurity and Infrastructure Security Agency (CISA). *CYBERSECURITY ADVISORY: PRC State-Sponsored Actors Compromise and Maintain Persistent Access to U.S. Critical Infrastructure*. 2024. Alert Code: AA24-038A. Accessed on: MAR. 21, 2024. Available at: <https://www.cisa.gov/news-events/cybersecurity-advisories/aa24-038a>.

10 WANG, Y.; LIU, H.; LI, Z.; SU, Z.; LI, J. Combating advanced persistent threats: Challenges and solutions. *IEEE Network*, p. 1–1, 2024.

11 Cybersecurity and Infrastructure Security Agency (CISA). *CYBERSECURITY ADVISORY: People's Republic of China State-Sponsored Cyber Actor Living off the Land to Evade Detection*. 2023. Alert Code: AA23-144A. Accessed on: DEC. 5, 2023. Available at: <https://www.cisa.gov/news-events/cybersecurity-advisories/aa23-144a>.

12 Cybersecurity and Infrastructure Security Agency (CISA). *Identifying and Mitigating Living Off the Land Techniques*. 2024. Accessed on: MAR. 21, 2024. Available at: <https://www.cisa.gov/resources-tools/resources/identifying-and-mitigating-living-land-techniques>.

13 NGAN, D. T. *Classification of Linux Commands in SSH Session by Risk Levels*. Thesis (Master) — University of Namur, Faculty of Computer Science, September 2020. Student thesis: Master in Computer Science with Professional focus in Data Science.

14 KOTSIANTIS, S.; KANELLOPOULOS, D.; PINTELAS, P. et al. Handling imbalanced datasets: A review. *GESTS international transactions on computer science and engineering*, v. 30, n. 1, p. 25–36, 2006.

15 GéRON, A. *Mãos à Obra: Aprendizado de Máquina com Scikit-Learn e TensorFlow: Conceitos, Ferramentas e Técnicas Para a Construção de Sistemas Inteligentes*. Alta Books, 2019. ISBN 978-8550803817. Available at: <https://www.amazon.com.br/M%C3%A3os-Obra-Aprendizado-Scikit-Learn-TensorFlow/dp/8550803812>.

16 KONG, Q.; SIAUW, T.; BAYEN, A. *Python programming and numerical methods: A guide for engineers and scientists*. [S.l.]: Academic Press, 2020.

17 MüLLER, A. C.; GUIDO, S. *Introduction to Machine Learning with Python: A Guide for Data Scientists*. O'Reilly Media, 2016. ISBN 978-1449369415. Available at: <https://www.oreilly.com/library/view/introduction-to-machine/9781449369880/>.

18 CONWAY, D.; WHITE, J. M. *Machine Learning for Hackers: Case Studies and Algorithms to Get You Started*. O'Reilly Media, 2012. ISBN 9781449303716. Available at: <https://books.google.com.br/books?id=2VY6Bbqd0F8C>.

19 BRUCE, P.; BRUCE, A. *Estatística Prática para Cientistas de Dados: 50 Conceitos Essenciais*. Alta Books, 2020. ISBN 978-8550806037. Available at: <https://www.amazon.com.br/Estat%C3%ADstica-Pr%C3%A1tica-Para-Cientistas-Dados/dp/855080603X/>.

20 ABUALHAJ, M. M.; ABU-SHAREHA, A. A.; SHAMBOUR, Q.; ALSAAIDAH, A.; AL-KHATIB, S. N.; ANBAR, M. Customized k-nearest neighbors' algorithm for malware detection. *International Journal of Data and Network Science*, Growing Science, v. 8, n. 1, 2024. ISSN 2561-8156.

21 VEZZETTI, E.; MARCOLIN, F. Minkowski distances for face recognition. In: *Advanced Methods and Applications in Computational Intelligence*. Bentham Science Publishers, 2015. p. 9–30. Available at: <https://www.eurekaselect.com/public/chapter/7522>.

22 ÇOLAKOğLU, H. B. A generalization of the minkowski distance and new definitions of the central conics. *Turkish Journal of Mathematics*, TÜBİTAK, v. 44, n. 1, p. 319–333, 2020. Available at: <https://www.researchgate.net/publication/338898480_A_generalization_of_the_Minkowski_distance_and_new_definitions_of_the_central_conics>.

23 BICEGO, M.; LOOG, M. Weighted k-nearest neighbor revisited. In: *Proceedings of the 2016 International Conference on Pattern Recognition (ICPR)*. IEEE, 2016. p. 1–6. Available at: <http://profs.sci.univr.it/~bicego/papers/2016_ICPR.pdf>.

24 JULIAN, D. *Designing Machine Learning Systems with Python*. Birmingham, UK: Packt Publishing, 2016. ISBN 978-1-78588-295-1. Available at: <https://books.google.com/books/about/Designing_Machine_Learning_Systems_with.html?id=193JDAAAQBAJ>.

25 SILVEIRA, G.; BULLOCK, B. *Machine Learning - Introdução à Classificação*. São Paulo, Brasil: Editora Casa do Código, 2020.

26 ALPAYDIN, E. *Introduction to Machine Learning, Fourth Edition (Adaptive Computation and Machine Learning series)*. 4th. ed. The MIT Press, 2020. 712 p. ISBN 978-0262043793. Available at: <https://www.amazon.com/-/pt/dp/0262043793/>.

27 AKILLI, A.; ATıL, H. Evaluation of normalization techniques on neural networks for the prediction of 305-day milk yield. 2020. ISSN 2717-8420.

28 HERWANTO, H. W.; HANDAYANI, A. N.; WIBAWA, A. P.; CHANDRIKA, K. L.; ARAI, K. Comparison of min-max, z-score and decimal scaling normalization for zoning feature extraction on javanese character recognition. In: *2021 7th International Conference on Electrical, Electronics and Information Engineering (ICEEIE)*. [S.l.: s.n.], 2021. p. 1–3.

29 SINSOMBOONTHONG, S. Performance comparison of new adjusted min-max with decimal scaling and statistical column normalization methods for artificial neural network classification. *Journal of Chemistry*, Hindawi Publishing Corporation, v. 2022, 2022. ISSN 1687-0425.

30 CASTILLO, J. R. del; MENDOZA-HURTADO, M.; ORTIZ-BOYER, D.; GARCíA-PEDRAJAS, N. Local-based k values for multi-label k-nearest neighbors rule. *Engineering Applications of Artificial Intelligence*, v. 116, p. 105487, 2022. ISSN 0952-1976. Available at: <https://www.sciencedirect.com/science/article/pii/S0952197622004778>.

31 GOLLAPUDI, S. *Practical Machine Learning: Tackle the Real-World Complexities of Modern Machine Learning with Innovative and Cutting-Edge Techniques*. Birmingham, UK: Packt Publishing, 2016. ISBN 978-1-78588-051-3. Available at: <https://www.packtpub.com/product/practical-machine-learning/9781785880513>.

32 LI, Y.; HERRERA-VIEDMA, E.; KOU, G.; MORENTE-MOLINERA, J. A. Z-number-valued rule-based decision trees. *Information Sciences*, Elsevier BV, v. 643, 2023. ISSN 1872-6291.

33 KIRK, M. *Thoughtful Machine Learning with Python: A Test-Driven Approach*. Sebastopol, CA: O'Reilly Media, 2015. ISBN 978-1-4919-1204-2. Available at: <https://www.oreilly.com/library/view/thoughtful-machine-learning/9781491912059/>.

34 ZHAO, H. Research on the application of improved decision tree algorithm based on information entropy in the financial management of colleges and universities. *International Journal of Advanced Computer Science and Applications (IJACSA)*, Science and Information Organization, v. 13, n. 12, 2022. ISSN 2158-107X.

35 LABER, E. S.; MURTINHO, L. Minimization of gini impurity: Np-completeness and approximation algorithm via connections with the k-means problem. *Electronic Notes in Theoretical Computer Science*, Elsevier BV, v. 346, 2019. ISSN 1571-0661.

36 GARRETA, R.; MONCECCHI, G. *Learning scikit-learn: Machine Learning in Python*. Birmingham, UK: Packt Publishing, 2013. Experience the benefits of machine learning techniques by applying them to real-world problems using Python and the open source scikit-learn library. ISBN 978-1783281930.

37 FU, X.; CHEN, Y.; YAN, J.; CHEN, Y.; XU, F. Bgrf: A broad granular random forest algorithm. *Journal of Intelligent & Fuzzy Systems*, IOS Press, v. 44, n. 5, 2023. ISSN 1875-8967.

38 ISONI, A. *Machine Learning for the Web: Explore the web and make smarter predictions using Python*. Birmingham, UK: Packt Publishing, 2016. Foreword by Davide Cervellin, Head of EU Analytics at eBay. ISBN 978-1785886188.

39 BRINK, H.; RICHARDS, J. W.; FETHEROLF, M. *Real-World Machine Learning*. Shelter Island, NY: Manning Publications, 2016.

40 SCHOBER, P.; VETTER, T. R. Logistic regression in medical research. *Anesthesia & Analgesia*, Lippincott Williams & Wilkins, v. 132, n. 2, 2021. ISSN 1526-7598.

41 BELL, J. *Machine Learning: Hands-On for Developers and Technical Professionals*. Indianapolis, IN: Wiley, 2014.

42 GUO, Y.; ZHAN, W.; LI, W. Application of support vector machine algorithm incorporating slime mould algorithm strategy in ancient glass classification. *Applied Sciences*, Multidisciplinary Digital Publishing Institute, v. 13, n. 6, p. 3718, 2023. ISSN 2076-3417.

43 BRANDEWINDER, M. *Machine Learning Projects for .NET Developers*. New York, NY: Apress, 2014. Build smarter applications by teaching them to learn from data. ISBN 978-1430267667.

44 SJARDIN, B.; MASSARON, L.; BOSCHETTI, A. *Large Scale Machine Learning with Python*. Birmingham, UK: Packt Publishing, 2016. Learn to build powerful machine learning models quickly and deploy large-scale predictive applications.

45 SWAMYNATHAN, M. *Mastering Machine Learning with Python in Six Steps: A Practical Implementation Guide to Predictive Data Analytics Using Python*. New York, NY: Apress, 2017. ISBN 978-1484228654.

46 SOARES, F. M.; SOUZA, A. M. *Neural Network Programming with Java*. Birmingham, UK: Packt Publishing, 2016. Unleash the power of neural networks by implementing professional Java code.

47 CAO, W. Evaluating the vocal music teaching using backpropagation neural network. *Mobile Information Systems*, v. 2022, p. 3843726, 2022.

48 BILSKI, J.; SMOLAG, J.; KOWALCZYK, B.; GRZANEK, K.; IZONIN, I. Fast computational approach to the levenberg-marquardt algorithm for training feedforward neural networks. *Journal of Artificial Intelligence and Soft Computing Research*, Sciendo, v. 13, n. 2, p. 45–61, 2023.

49 KALUžA, B. *Machine Learning in Java: Design, build, and deploy your own machine learning applications by leveraging key Java machine learning libraries*. Birmingham, UK: Packt Publishing, 2016. ISBN 978-1785884035.

50 MIHALCEA, R.; LIU, H.; LIEBERMAN, H. NLP (Natural Language Processing) for NLP (Natural Language Programming). In: *Proceedings of Springer Science+Business Media*. [S.l.]: Springer, 2006. ISSN 1611-3349.

51 VERMA, P.; GOYAL, A.; GIGRAS, Y. Email phishing: text classification using natural language processing. *Institute of Advanced Engineering and Science (IAES)*, v. 1, n. 1, p. 1–12, 2020. ISSN 2722-323X.

52 ENRíQUEZ, F.; TROYANO, J. A.; LóPEZ-SOLAZ, T. An approach to the use of word embeddings in an opinion classification task. *Expert Systems with Applications*, Elsevier BV, v. 66, p. —, 2016. ISSN 1873-6793.

53 NIXON, R.; FAUSTINO, H.; CASTRO-GUTIéRREZ, E. Automatic cyberbullying detection in spanish-language social networks using sentiment analysis techniques. *International Journal of Advanced Computer Science and Applications*, Science and Information Organization, v. 9, n. 7, 2018. ISSN 2158-107X.

54 KHOMSAH, S.; RAMADHANI, R. D.; WIJAYA, S. The accuracy comparison between word2vec and fasttext on sentiment analysis of hotel reviews. *Jurnal RESTI (Rekayasa Sistem dan Teknologi Informasi)*, Ikatan Ahli Informatika Indonesia, v. 6, n. 3, 2022. ISSN 2580-0760.

55 WANG, Z.; WANG, D.; LI, Q. Keyword extraction from scientific research projects based on srp-tf-idf. *Institution of Engineering and Technology*, v. 30, n. 4, 2021. ISSN 2075-5597.

56 NI'MAH, A. T.; ARIFIN, A. Z. Perbandingan metode term weighting terhadap hasil klasifikasi teks pada dataset terjemahan kitab hadis. *Rekayasa*, Institut Teknologi Sepuluh Nopember, v. 13, n. 2, p. 172–180, 2020. Available at: <https://doi.org/10.21107/rekayasa.v13i2.6412>.

57 LAU, J. H.; BALDWIN, T. An empirical evaluation of doc2vec with practical insights into document embedding generation. *arXiv e-prints*, p. arXiv:1607.05368, jul. 2016. Available at: <https://ui.adsabs.harvard.edu/abs/2016arXiv160705368L>.

58 CHEN, Q.; SOKOLOVA, M. Specialists, scientists, and sentiments: Word2vec and doc2vec in analysis of scientific and medical texts. *Springer Nature*, v. 2, n. 5, 2021. ISSN 2662-995X.

59 RASCHKA, S. *Python Machine Learning: Unlock deeper insights into machine learning with this vital guide to cutting-edge predictive analytics*. Birmingham, UK: Packt Publishing, 2015.

60 MUHAMMAD, P. F.; KUSUMANINGRUM, R.; WIBOWO, A. Sentiment analysis using word2vec and long short-term memory (lstm) for indonesian hotel reviews. *Procedia Computer Science*, Elsevier BV, v. 179, p. 582–589, 2021. ISSN 1877-0509.

61 LE, Q.; MIKOLOV, T. Distributed representations of sentences and documents. *arXiv preprint arXiv:1309.4168*, 2013. Available at: <http://arxiv.org/pdf/1309.4168v1>.

62 LE, Q. V.; MIKOLOV, T. Distributed representations of sentences and documents. *arXiv preprint arXiv:1405.4053*, 2014.

63 ZHENG, M.; WANG, F.; HU, X.; MIAO, Y.; CAO, H.; TANG, M. A method for analyzing the performance impact of imbalanced binary data on machine learning models. *Axioms*, Multidisciplinary Digital Publishing Institute, v. 11, n. 11, p. 607, 2022. ISSN 2075-1680.

64 BAGUI, S.; LI, K. Resampling imbalanced data for network intrusion detection datasets. *Journal of Big Data*, Springer, v. 8, n. 6, p. 1–41, 2021. Available at: <https://doi.org/10.1186/s40537-020-00390-x>.

65 CHAWLA, N. V.; BOWYER, K. W.; HALL, L. O.; KEGELMEYER, W. P. Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, AI Access Foundation, v. 16, p. 321–357, 2002.

66 KUBAT, M. Addressing the curse of imbalanced training sets: One-sided selection. *Fourteenth International Conference on Machine Learning*, 06 2000.

67 JIANG, K.; WANG, W.; WANG, A.; WU, H. Network intrusion detection combined hybrid sampling with deep hierarchical network. *IEEE Access*, Institute of Electrical and Electronics Engineers, v. 8, p. 2973730, 2020. ISSN 2169-3536.

68 MQADI, N. M.; NAICKER, N.; ADELIYI, T. T. Solving misclassification of the credit card imbalance problem using near miss. *Hindawi Publishing Corporation*, v. 2021, p. 7194728, 2021. ISSN 1563-5147.

69 PATRO, M.; PATRA, M. R. A novel approach to compute confusion matrix for classification of n-class attributes with feature selection. *The Machine Learning and Artificial Intelligence Journal*, The Author(s), v. 3, n. 2, p. 1108, 2015. ISSN 2054-7390.

70 HARRISON, M. *Machine Learning: Guia de Referência Rápida*. [S.l.]: Novatec, 2024. Trabalhando com dados estruturados em Python.

71 COOK, D. *Practical Machine Learning with H2O: Powerful, Scalable Techniques for AI and Deep Learning*. Sebastopol, CA: O'Reilly Media, 2016.

72 CHICCO, D.; JURMAN, G. The matthews correlation coefficient (mcc) should replace the roc auc as the standard metric for assessing binary classification. *BioMed Central*, BioMed Central, v. 16, n. 1, 2023. ISSN 1756-0381.

73 PARASA, S.; REPICI, A.; BERZIN, T. M.; LEGGETT, C. L.; GROSS, S. A.; SHARMA, P. Framework and metrics for the clinical use and implementation of artificial intelligence algorithms into endoscopy practice: recommendations from the american society for gastrointestinal endoscopy artificial intelligence task force. *Gastrointestinal Endoscopy*, Elsevier BV, v. 97, n. 5, 2023. ISSN 1097-6779.

74 CHICCO, D.; JURMAN, G. The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. *BMC genomics*, BioMed Central, v. 21, n. 1, p. 1–13, 2020.

75 CHICCO, D.; TÖTSCH, N.; JURMAN, G. The matthews correlation coefficient (mcc) is more reliable than balanced accuracy, bookmaker informedness, and markedness in two-class confusion matrix evaluation. *BioData mining*, BioMed Central, v. 14, n. 1, p. 1–22, 2021.

76 SYARIF, I.; PRUGEL-BENNETT, A.; WILLS, G. Svm parameter optimization using grid search and genetic algorithm to improve classification performance. *TELKOMNIKA*, Universitas Ahmad Dahlan, v. 14, n. 4, p. 1502–1509, 2016. ISSN 1693-6930.

77 REIF, M.; SHAFAIT, F.; DENGEL, A. Meta-learning for evolutionary parameter optimization of classifiers. *Machine Learning*, Springer Science+Business Media, v. 87, n. 3, 2012. ISSN 1573-0565. Available at: <https://doi.org/10.1007/s10994-012-5286-7>.

78 JIANG, X.; XU, C. Deep learning and machine learning with grid search to predict later occurrence of breast cancer metastasis using clinical data. *Journal of Clinical Medicine*, Multidisciplinary Digital Publishing Institute, v. 11, n. 19, 2022. ISSN 2077-0383. Available at: <https://doi.org/10.3390/jcm11195772>.

79 CHUMACHENKO, K. Machine learning methods for malware detection and classification. Kaakkois-Suomen ammattikorkeakoulu, 2017.

80 RATHORE, H.; AGARWAL, S.; SAHAY, S. K.; SEWAK, M. Malware detection using machine learning and deep learning. In: SPRINGER. *Big Data Analytics: 6th International Conference, BDA 2018, Warangal, India, December 18–21, 2018, Proceedings 6*. [S.l.], 2018. p. 402–411.

81 FINDER, I.; SHEETRIT, E.; NISSIM, N. A time-interval-based active learning framework for enhanced PE malware acquisition and detection. *Comput. Secur.*, v. 121, p. 102838, 2022.

82 BOROS, T.; COTAIE, A.; STAN, A.; VIKRAMJEET, K.; MALIK, V.; DAVIDSON, J. *Machine Learning and Feature Engineering for Detecting Living off the Land Attacks*. 2022.

83 ONGUN, T.; STOKES, J. W.; OR, J. B.; TIAN, K.; TAJADDODIANFAR, F.; NEIL, J.; SEIFERT, C.; OPREA, A.; PLATT, J. C. Living-off-the-land command detection using active learning. In: *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*. New York, NY, USA: Association for Computing Machinery, 2021. (RAID '21), p. 442–455. ISBN 9781450390583. Available at: <https://doi.org/10.1145/3471621.3471858>.

84 DEMMER, S. Detecting "living-off-the-land" attackertechniques in microsoft windows. *Tese de Mestrado, Universidade de Ciências Aplicadas de St. Pölten, Programa de Mestrado em Segurança da Informação, 2019*, 2019. Available at: <https://phaidra.fhstp.ac.at/detail/o:4260?mode=full&tab=82>.

85 ROTH, F. *Sigma project*. 2024. Accessed on: Out. 21, 2024. Available at: <https://github.com/Neo23x0/sigma>.

86 HENDLER, D.; KELS, S.; RUBIN, A. Detecting malicious powershell commands using deep neural networks. In: *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2018. (ASIACCS '18), p. 187–197. ISBN 9781450355766. Available at: <https://doi.org/10.1145/3196494.3196511>.

87 DEWILDE, B. *Textacy Documentation*. [S.l.], 2023. Available at: <https://textacy.readthedocs.io/en/latest/>.

88 JURAFSKY, D.; MARTIN, J. H. *Speech and Language Processing (3rd ed. draft)*. [s.n.], 2023. Available at: <https://web.stanford.edu/~jurafsky/slp3/>.

89 Snorkel Development Team. *Snorkel: A System for Quickly Generating Training Data with Weak Supervision*. 2023. <https://www.snorkel.org/get-started/>.

90 TSAI, M.-H.; LIN, C.-C.; HE, Z.-G.; YANG, W.-C.; LEI, C.-L. Powerdp: De-obfuscating and profiling malicious powershell commands with multi-label classifiers. *IEEE Access*, IEEE, v. 11, p. 256–270, 2023. ISSN 2169-3536. Available at: <https://ieeexplore.ieee.org/document/9999441/>.

91 KOLLI, S.; BALAKESAVAREDDY, P.; SARAVANAN, D. Neural network based obfuscated malware detection. In: *2021 International Conference on System, Computation, Automation and Networking (ICSCAN)*. [S.l.: s.n.], 2021. p. 1–5.

92 KHAN, L. P. Obfuscated malware detection using artificial neural network (ann). In: *2023 Fifth International Conference on Electrical, Computer and Communication Technologies (ICECCT)*. [S.l.: s.n.], 2023. p. 1–5.

93 HASAN, S. M. R.; DHAKAL, A. Obfuscated malware detection: Investigating real-world scenarios through memory analysis. In: *2023 IEEE International Conference on Telecommunications and Photonics (ICTP)*. [S.l.: s.n.], 2023. p. 01–05.

94 WEISS, G. Foundations of imbalanced learning. In: ____. [S.l.: s.n.], 2013. p. 13–41. ISBN 9781118074626.

95 HARRINGTON, P. *Machine Learning in Action*. Shelter Island: Manning Publications, 2012.

96 ALAMSYAH, A. R.; RAHMA, S.; BELINDA, N. S.; SETIAWAN, A. Smote and nearmiss methods for disease classification with unbalanced data case study: Ifls 5. In: . [S.l.: s.n.], 2022. v. 2021.

97 JOSEFSSON, S. RFC, *The Base16, Base32, and Base64 Data Encodings*. RFC Editor, 2006. Internet Requests for Comments. Section 4. Available at: <https://datatracker.ietf.org/doc/html/rfc4648>.

98 The Linux Documentation Project. *Advanced Bash-Scripting Guide: Chapter 12. Command Substitution*. 2008. <https://tldp.org/LDP/abs/html/commandsub.html>. Accessed on: MAY 5, 2024.

99 GNU coreutils. *echo - display a line of text*. [S.l.], 2023. Accessed on: MAY 5, 2024. Available at: <https://www.man7.org/linux/man-pages/man1/echo.1.html>.

100 YERGEAU, F. RFC, *UTF-8, a transformation format of ISO 10646*. RFC Editor, 2003. Internet Requests for Comments. Available at: <https://datatracker.ietf.org/doc/html/rfc3629>.

101 HORNIG, C. RFC, *A Standard for the Transmission of IP Datagrams over Ethernet Networks*. RFC Editor, 1984. Internet Requests for Comments. Available at: <https://datatracker.ietf.org/doc/html/rfc894>.

I.1 APPENDIX A - PLAIN NORMALIZATION

```python
def replace_quoted_string(text):
    # Replaces everything in quotes with _STRING_
    return re.sub(r"\"(.+?)\"|\'(.+?)\'", "_STRING_", text)

def replace_file_path(text):
    # Replace file name or directory name with _PATH_
    return re.sub(r"((?<= )[^ ]*/)([^/ ]*)", r"_PATH_/\2", text) # Modified to
        only replace the directory not the final file

def replace_env_variables(text):
    # Replace $String with $_ENV and String= with _ENV_=.
    text = re.sub(r"\$([A-Za-z_][A-Za-z0-9_]*)", r"$_ENV", text)
    text = re.sub(r"([A-Za-z_][A-Za-z0-9_]*)=", r"_ENV_=", text)
    return text

def replace_ip_address(text):
    # Replace IPv4 with _IP_
    return re.sub("[0-9]+(?:\.[0-9]+){3}", "_IP_", text)

def refine_pipe(text):
    # Make clear seperation between different commands in pipe.
    return text.replace("|", " | ")

def normalize(line):
    # convert to lowercase
    line = line.lower()
    # Replace $String with $_ENV and String= with _ENV_=
    line = replace_env_variables(line)
    # normalize quoted mark
    line = quotation_marks(line) # TEXTACY's function
    # refine pipe operator
    line = refine_pipe(line)
    # replace quoted text by _STRING_
    line = replace_quoted_string(line)
    # replace IP adress by '_IP_' token
    line = replace_ip_address(line)
    # replace file path by '_PATH_' token
    line = replace_file_path(line)
    # replace email by '_EMAIL_' token
    line = emails(line, "_EMAIL_") #TEXTACY's function
    # replace URL by '_URL_' token
    line = urls(line, "_URL_") #TEXTACY's function
```

```
    return line
```

Code 1: First Normalization Code

I.2 APPENDIX B - DEOBFUSCATION NORMALIZATION

```
def replace_quoted_string(text):
    # Removes quotes but keeps the content between quotes.
    return re.sub(r'"(.*?)"|\'(.*?)\'', lambda m: m.group(1) if m.group(1) is not None else m.group(2), text)


def replace_file_path(text):
    # Replace file name or directory name with _PATH_
    # return re.sub(r"(~)*(/[^/ ]*)+", "_PATH_", text) # ((?:[^/]*/)*)(.*)
    return re.sub(r"((?<= )[^ ]*/)([^/ ]*)", r"_PATH_/\2", text) # Modified to replace only the directory, not the final file

def replace_env_variables(text):
    #Replace $String with $_ENV and String= with _ENV_=.
    text = re.sub(r"\$([A-Za-z_][A-Za-z0-9_]*)", r"$_ENV", text)
    text = re.sub(r"([A-Za-z_][A-Za-z0-9_]*)=", r"_ENV_=", text)
    return text

def replace_ip_address(text):
    #Replace IPv4 with _IP_
    return re.sub("[0-9]+(?:\.[0-9]+){3}", "_IP_", text)

def refine_pipe(text):
    """Make a clear separation between different commands in a pipe.
    E.g.: cmd1|cmd2 -> cmd1 | cmd2
    """
    return text.replace("|", " | ")

def replace_dollar_parentheses(text):
    # Replaces the pattern $(STRING) with ' STRING ', inserting spaces where $( and ) were.
    return re.sub(r'\$\((.*?)\)', r' \1 ', text)

def replace_backticks(text):
    # Replaces the ` character with a space.
    return text.replace('`', ' ')

def hex_to_char(match):
    try:
        return bytes.fromhex(match.group(1)).decode('utf-8')
    except UnicodeDecodeError:
        return bytes.fromhex(match.group(1)).decode('latin-1')
```

```
    # Converts a hexadecimal sequence to the corresponding character.

def unicode_to_char(match):
    # Converts a Unicode sequence to the corresponding character.
    # The substring 2: is used to skip the '\u'.
    return chr(int(match.group(1), 16))

def octal_to_char(match):
    try:
        return chr(int(match.group(1), 8))
    except ValueError:
        return match.group(0)

def decode_escapes(text):
    # Decodes hexadecimal sequences.
    text = re.sub(r'\\x([0-9A-Fa-f]{2})', hex_to_char, text)
    # Decodes Unicode sequences.
    text = re.sub(r'\\u([0-9A-Fa-f]{4})', unicode_to_char, text)
    # Decodes octal sequences - Assumes the octal is preceded by a \ and followed
        by 1 to 3 digits [0-7].
    text = re.sub(r'\\([0-3]?[0-7]{2,3})', octal_to_char, text)

    return text

def decode_base64_if_needed(s):
    # Regular expression to identify potential base64 substrings.
    base64_pattern = re.compile(r'([A-Za-z0-9+/=]{20,}={0,2})')

    def decode_base64(match):
        try:
            base64_str = match.group()
            # Ensure the string length is valid for base64.
            if len(base64_str) % 4 != 0:
                base64_str += '=' * (4 - len(base64_str) % 4)
            decoded_bytes = base64.b64decode(base64_str, validate=True)
            return decoded_bytes.decode('utf-8')
        except (binascii.Error, ValueError, UnicodeDecodeError):
            return match.group()  # Return as is if decoding fails

    # Substitute base64 substrings with their decoded versions.
    decoded_str = base64_pattern.sub(decode_base64, s)
    return decoded_str, decoded_str != s

def replace_env_concatenation(text):
    """
    Replaces environment variable concatenations with 'CONCATENATION'.
    A concatenation pattern is identified by sequences of $VARNAME.
    """
    return re.sub(r'(\$\w+ \s*\$\w+)+', 'CONCATENATION', text)

def re_normalize(line):
    ### NEW ###
```

```
        # First, check and decode base64 if necessary.
        # print("Original Line:", line)
        line, was_base64 = decode_base64_if_needed(line)
        # print("After Base64 Decode:", line, was_base64)

        # Encoding
        line = decode_escapes(line)
        # print("After decode_escapes:", line)

        # Apply the new replacement functions.
        line = replace_dollar_parentheses(line)
        # print("After replace_dollar_parentheses:", line)
        line = replace_backticks(line)
        # print("After replace_backticks:", line)

        # CONCATENATION
        line = replace_env_concatenation(line)
        # print("After replace_env_concatenation:", line)

        # Convert to lowercase.
        line = line.lower()
        # print("After convert to lowercase:", line)

        # Replace $String with $_ENV and String= with _ENV_=.
        line = replace_env_variables(line)
        # print("After replace_env_variables:", line)

        # Normalize quoted marks.
        line = quotation_marks(line) # Function from TEXTACY
        # print("After quotation_marks:", line)

        # Refine pipe operator.
        line = refine_pipe(line)
        # print("After refine_pipe:", line)

        # Replace quoted text by '_STRING_'.
        line = replace_quoted_string(line)
        # print("After replace_quoted_string:", line)

        # Replace IP address by '_IP_' token.
        line = replace_ip_address(line)
        # print("After replace_ip_address:", line)

        # Replace file path by '_PATH_' token.
        line = replace_file_path(line)
        # print("After replace_file_path:", line)

        # Replace email by '_EMAIL_' token.
        line = emails(line, "_EMAIL_") # Function from TEXTACY
        # print("After emails:", line)

        # Replace URL by '_URL_' token.
```

```
    line = urls(line, "_URL_") # Function from TEXTACY
    # print("After urls:", line)

    return line
```

Code 2: First Normalization Code

I.3 APPENDIX C - OBFUSCATION AND TRAINING ROUTINE

```
print ("####################################")
print ("Reading the GFTOBINS")

## 01 - Reading the GTFOBINS

gtfo_df = pd.read_json('https://gtfobins.github.io/gtfobins.json')

gtfo_list = []
for function in gtfo_df.T.functions:
    for keys in function:
        for i in function[keys]:
            for key in i:
                if key == 'code':
                    gtfo_list.append(i[key])

# Splitting the commands by \n and creating a list of lists
gfto_split = []
for x in gtfo_list:
    gfto_split.append(x.split('\n'))

# Normalizing the commands with the normalize() function
gtfo_norm = []
for x in gfto_split:
    a = []
    for y in x:
        a.append(re_normalize(y))
    gtfo_norm.append(a)

# Removing empty items from the list
for cmd in gtfo_norm:
    for x in cmd:
        if x == '':
            cmd.remove(x)

gtfo_norm[1:5]


## 02 - Reading the Benign

```

```
print ("####################################")
print ("Reading the Benign")

import os
import re
import itertools

data_path = "./Datasets/"
bash_logs_path = os.path.join(data_path, "bash_logs")

# Read bash log data
bash_file_names = os.listdir(bash_logs_path)
benign_logs = []

for file in bash_file_names:
    file_path = os.path.join(bash_logs_path, file)
    with open(file_path, 'rb') as f:  # Opening in binary mode
        # Reading the content and decoding line by line
        content = f.read().replace(b"&&", b"\n").replace(b";", b"\n").splitlines()
        for line in content:
            try:
                # Decode each line individually
                decoded_line = line.decode('utf-8', errors='ignore')
                # Normalize the decoded line
                normalized_line = re_normalize(decoded_line)
                benign_logs.append(normalized_line)
            except UnicodeDecodeError as e:
                print(f"Error decoding line: {e}")
                continue

# Remove empty lines
benign_unique = set(benign_logs)
benign_unique.discard("")


## 03 - Transforming Dataset
print ("####################################")
print ("Transforming Dataset")

import base64

def convert_command_to_hex(s):
    """Converts the string to hexadecimal."""
    return ''.join(f'\\x{ord(c):02x}' for c in s)

def convert_command_to_octal(s):
    """Converts the string to octal with the prefix \0."""
    return ''.join(f'\\0{ord(c):03o}' for c in s)

def convert_command_to_unicode(s):
    """Converts the string to unicode."""
    return ''.join(f'\\u{ord(c):04x}' for c in s)
```

```python

def transform_strings_to_base64_bash(strings):
    """
    Transforms each string to base64 and creates a new string with the structure
    "echo [STRING_BASE64] | base64 -d | bash".

    Args:
    strings (list of str): List of strings to be transformed.

    Returns:
    list of str: List with the transformed strings.
    """
    new_strings = []

    for s in strings:
        # Encode the string to base64
        encoded_bytes = base64.b64encode(s.encode('utf-8'))
        encoded_str = encoded_bytes.decode('utf-8')

        # Create the new string with the desired structure
        new_str = f"echo {encoded_str} | base64 -d | bash"

        # Add the new string to the list
        new_strings.append(new_str)

    return new_strings


def generate_alternatives(strings):
    """
    For each string in the list, generates two new items:
    one with the content between $() and another with the content between ``.

    Args:
    strings (list of str): List of strings to be transformed.

    Returns:
    list of str: List with the transformed strings.
    """
    new_strings = []

    for s in strings:
        # Create the new strings with the desired structures
        new_str_dollar = f"$({s})"
        new_str_backticks = f"`{s}`"

        # Add the new strings to the list
        new_strings.append(new_str_dollar)
        new_strings.append(new_str_backticks)

    return new_strings

```

```python
def generate_encoded_commands(strings):
    """
    For each string in the list, generates three new items:
    one with the command converted to hexadecimal,
    one to octal, and one to unicode.

    Args:
    strings (list of str): List of strings to be transformed.

    Returns:
    list of str: List with the transformed strings.
    """
    new_strings = []

    for s in strings:
        # Convert command to hexadecimal
        hex_str = f'echo -e "{convert_command_to_hex(s)}" | sh'

        # Convert command to octal
        octal_str = f'echo -e "{convert_command_to_octal(s)}" | sh'

        # Convert command to unicode
        unicode_str = f'echo -e "{convert_command_to_unicode(s)}" | sh'

        # Add the new strings to the list
        new_strings.append(hex_str)
        new_strings.append(octal_str)
        new_strings.append(unicode_str)

    return new_strings


def combine_all_transformations(strings):
    """
    Combines all string transformations into a single list.

    Args:
    strings (list of str): List of strings to be transformed.

    Returns:
    list of str: List with all transformed strings.
    """
    combined_list = []

    # Apply base64 transformation
    combined_list.extend(transform_strings_to_base64_bash(strings))

    # Apply alternatives generation
    combined_list.extend(generate_alternatives(strings))

    # Apply encoded commands generation
    combined_list.extend(generate_encoded_commands(strings))
```

```
196
197     return combined_list
198
199 # Test the function with a list of strings
200 test_strings = cmds_flat
201 transformed_list = combine_all_transformations(test_strings)
202 transformed_list2 = combine_all_transformations(benign_unique)
203
204
205 print ("LOLBINS transformed:  {0}".format(len(transformed_list)))
206 print ("Benign transformed:  {0}".format(len(transformed_list2)))
207
208
209 ## 04 - Normalizing the transformed Lists
210 print ("####################################")
211 print ("Normalizing the transformed Lists")
212
213 def apply_normalize_to_transformed_list(transformed_list):
214     """
215     Applies the normalize() function to each item in the transformed list.
216
217     Args:
218     transformed_list (list of str): List of transformed strings.
219
220     Returns:
221     list of str: List with normalized strings.
222     """
223     normalized_list = [re_normalize(item) for item in transformed_list]
224     return normalized_list
225
226
227 normalized_list = apply_normalize_to_transformed_list(transformed_list)
228 normalized_list2 = apply_normalize_to_transformed_list(transformed_list2)
229
230
231 # 05 - Creating the New Dataset
232 print ("####################################")
233 print ("Creating the New Dataset")
234
235 # WINDOW 03 commands
236 def create_n_command_by_sliding_window(session_cmds, labels, window_size=3):
237     cmds_flat = []
238     labels_flat = []
239
240     for sess_idx, cmds in enumerate(session_cmds):
241         for i in range(len(cmds) + 1):
242             start_idx = max(0, i - window_size)
243             if start_idx == i:
244                 continue
245             cmds_flat.append(" ".join(cmds[start_idx:i]))
246             labels_flat.append(labels[sess_idx])
247
```

```
248     return cmds_flat, labels_flat
249 
250 WINDOW_SIZE = 3
251 
252 normalized_list2_labels = [0] * len(normalized_list2)
253 norm_labels = [1] * len(normalized_list)
254 benign_labels = [0] * len(benign_logs)
255 GTFOBins_labels = [1] * len(gtfo_norm)
256 
257 session_cmds = benign_logs + gtfo_norm + normalized_list + normalized_list2
258 labels = benign_labels + GTFOBins_labels + norm_labels + normalized_list2_labels
259 
260 print(len(session_cmds))
261 print(len(labels))
262 
263 
264 cmds_flat, labels_flat = create_n_command_by_sliding_window(
265     session_cmds, labels, window_size=WINDOW_SIZE
266 )
267 
268 print("N-commands", len(cmds_flat), len(labels_flat))
269 
270 
271 # create tagged corpus: {[list of token], [tag]}
272 
273 from gensim.models import doc2vec
274 
275 all_corpus = []
276 
277 for cmd, lbl in zip(cmds_flat, labels_flat):
278     tokens = cmd.split()
279     all_corpus.append(doc2vec.TaggedDocument(tokens, str(lbl)))
280 
281 # build Doc2Vec model
282 print ("build Doc2Vec model")
283 VECTOR_SIZE = 64
284 
285 model = doc2vec.Doc2Vec(
286     vector_size=VECTOR_SIZE,
287     min_count=2,
288 )
289 
290 print(len(all_corpus))
291 model.build_vocab(all_corpus)
292 
293 # train model
294 
295 model.train(all_corpus, total_examples=model.corpus_count, epochs=model.epochs)
296 
297 # build feature
298 
299 def extract_features(tagged_corpus, model):
```

```
    X = []
    y = []

    for words, tags in tagged_corpus:
        X.append(model.infer_vector(words))
        y.append(float(tags))

    return X, y

X_train_features, y = extract_features(all_corpus, model)

print(len(X_train_features), len(y))

def create_balanced_dataset(X, Y):
    '''
    Function to apply various data balancing algorithms on unbalanced datasets:
    '''

    # Random Oversampler
    ros = RandomOverSampler(random_state=32)
    X_ros_res, y_ros_res = ros.fit_resample(X, Y)

    return X_ros_res, y_ros_res

X_ros_res, y_ros_res = create_balanced_dataset(X_train_features, y)


## 06 - RF Training
print ("####################################")
print ("RF Training")

best_params = {
    'criterion': 'entropy',
    'min_samples_leaf': 5,
    'min_samples_split': 5,
    'n_estimators': 100
}

rf_model = RandomForestClassifier(**best_params)

def fit_predict(model, X, Y):
    train_cmds, test_cmds, train_labels, test_labels = train_test_split(
        X, Y, test_size=0.3, random_state=42
    )

    # Training the model
    model.fit(train_cmds, train_labels)

    # Predicting the results
    y_pred = model.predict(test_cmds)

    # Calculating metrics
```

```
    acc = accuracy_score(test_labels, y_pred) * 100
    f1 = f1_score(test_labels, y_pred) * 100
    mcc = matthews_corrcoef(test_labels, y_pred) * 100
    final_score = (acc + 2 * f1 + mcc) / 4  # Final score

    # Returning metrics
    print("Accuracy: ", acc)
    print("F1-Score: ", f1)
    print("MCC: ", mcc)
    print("Final Score: ", final_score)

    return {"Accuracy": acc, "F1-Score": f1, "MCC": mcc, "Final Score": final_
        score}


fit_predict(rf_model, X_ros_res, y_ros_res)
```

Code 3: First Normalization Code

I.4 APPENDIX C - RESULTS FROM BOW TRAINING

Table 1: Results from BOW

Accuracy	F1-Score	MCC	Tecnica	Modelo	Metodo
86.236766	86.928702	73.563865	Random Undersampler	KNN	BoW
87.776708	86.603376	75.979806	NearMiss	KNN	BoW
98.817020	61.840121	61.362812	OneSidedSelection	KNN	BoW
91.555901	91.153967	83.396267	Random Oversampler	KNN	BoW
91.262813	91.840388	83.467727	SMOTE	KNN	BoW
89.749759	88.900469	79.791425	Random Undersampler	DT	BoW
88.594803	87.559055	77.562369	NearMiss	DT	BoW
98.941708	61.442362	62.209619	OneSidedSelection	DT	BoW
91.959094	91.606096	84.160301	Random Oversampler	DT	BoW
91.894140	91.490724	84.103609	SMOTE	DT	BoW
90.038499	89.246753	80.333341	Random Undersampler	RF	BoW
88.787295	87.794657	77.920958	NearMiss	RF	BoW
98.957294	62.096317	62.848401	OneSidedSelection	RF	BoW
91.991572	91.637164	84.229661	Random Oversampler	RF	BoW
91.921864	91.517642	84.162908	SMOTE	RF	BoW
88.931665	87.817797	78.385989	Random Undersampler	RL	BoW
88.306064	87.337155	76.880304	NearMiss	RL	BoW
98.580112	34.882059	40.041677	OneSidedSelection	RL	BoW
89.385466	88.855251	79.067599	Random Oversampler	RL	BoW
90.272651	89.783524	80.853705	SMOTE	RL	BoW
87.872955	86.736842	76.142482	Random Undersampler	SVC	BoW
88.739172	87.710084	77.861748	NearMiss	SVC	BoW
98.561409	23.781998	35.296420	OneSidedSelection	SVC	BoW
89.061485	88.493555	78.441824	Random Oversampler	SVC	BoW
89.973226	90.641312	80.875831	SMOTE	SVC	BoW
89.172281	88.101534	78.846109	Random Undersampler	NN	BoW
89.124158	88.204593	78.550472	NearMiss	NN	BoW
98.896509	64.741036	64.271067	OneSidedSelection	NN	BoW
92.024841	91.665563	84.305972	Random Oversampler	NN	BoW
91.784034	92.320109	84.496137	SMOTE	NN	BoW

I.5 APPENDIX D - RESULTS FROM DOC2VEC TRAINING

Table 2: Results from Doc2Vec

Accuracy	F1-Score	MCC	Tecnica	Modelo	Metodo
85.686747	86.166744	72.302557	Random Undersampler	KNN	DOC2VEC
94.313253	93.703308	89.141766	NearMiss	KNN	DOC2VEC
99.033386	56.439127	60.802520	OneSidedSelection	KNN	DOC2VEC
98.726256	98.737556	97.484353	Random Oversampler	KNN	DOC2VEC
91.763438	92.363508	84.692518	SMOTE	KNN	DOC2VEC
78.265060	77.881314	56.597416	Random Undersampler	DT	DOC2VEC
95.180723	95.000000	90.351674	NearMiss	DT	DOC2VEC
98.008995	39.140811	38.205148	OneSidedSelection	DT	DOC2VEC
99.354415	99.356133	98.717102	Random Oversampler	DT	DOC2VEC
96.374424	96.412475	92.787978	SMOTE	DT	DOC2VEC
89.204819	88.405797	78.457161	Random Undersampler	RF	DOC2VEC
98.168675	98.107570	96.345225	NearMiss	RF	DOC2VEC
98.760111	32.597623	43.664500	OneSidedSelection	RF	DOC2VEC
99.999208	99.999205	99.998416	Random Oversampler	RF	DOC2VEC
99.809097	99.808339	99.618193	SMOTE	RF	DOC2VEC
88.481928	87.787430	76.939764	Random Undersampler	RL	DOC2VEC
96.674699	96.495683	93.355875	NearMiss	RL	DOC2VEC
98.944377	49.627422	55.763606	OneSidedSelection	RL	DOC2VEC
90.812883	90.562291	81.707105	Random Oversampler	RL	DOC2VEC
91.974145	91.821117	83.983672	SMOTE	RL	DOC2VEC
89.638554	88.968702	79.277942	Random Undersampler	SVC	DOC2VEC
97.253012	97.139990	94.497476	NearMiss	SVC	DOC2VEC
99.097411	60.027663	64.038837	OneSidedSelection	SVC	DOC2VEC
91.559069	91.298241	83.228831	Random Oversampler	SVC	DOC2VEC
92.653792	92.499070	85.353941	SMOTE	SVC	DOC2VEC
91.421687	91.231527	82.896695	Random Undersampler	NN	DOC2VEC
97.638554	97.541395	95.269767	NearMiss	NN	DOC2VEC
99.270745	76.473552	76.106091	OneSidedSelection	NN	DOC2VEC
99.311640	99.311776	98.626432	Random Oversampler	NN	DOC2VEC
99.647502	99.647024	99.296182	SMOTE	NN	DOC2VEC

I.6 APPENDIX E - RESULTS FROM TF-IDF TRAINING

Table 3: Results from TFIDF

Accuracy	F1-Score	MCC	Tecnica	Modelo	Metodo
91.710843	91.706847	83.708577	Random Undersampler	KNN	TFIDF
92.048193	91.585926	84.089564	NearMiss	KNN	TFIDF
98.971209	65.517241	65.038036	OneSidedSelection	KNN	TFIDF
94.153293	94.100250	88.310742	Random Oversampler	KNN	TFIDF
94.059822	93.901120	88.204764	SMOTE	KNN	TFIDF
91.759036	91.914894	84.159502	Random Undersampler	DT	TFIDF
92.096386	91.606960	84.201017	NearMiss	DT	TFIDF
99.033560	69.276511	68.797029	OneSidedSelection	DT	TFIDF
94.549358	94.754096	89.430359	Random Oversampler	DT	TFIDF
94.859080	95.053881	90.057478	SMOTE	DT	TFIDF
93.590361	93.621103	87.557201	Random Undersampler	RF	TFIDF
92.530120	92.103923	85.052937	NearMiss	RF	TFIDF
99.095911	70.408163	69.953710	OneSidedSelection	RF	TFIDF
94.606391	94.806136	89.536021	Random Oversampler	RF	TFIDF
94.974731	95.163307	90.283357	SMOTE	RF	TFIDF
91.807229	91.500000	83.595759	Random Undersampler	RL	TFIDF
91.518072	91.020408	83.027664	NearMiss	RL	TFIDF
98.689071	51.638873	51.506664	OneSidedSelection	RL	TFIDF
90.968141	90.920963	81.936155	Random Oversampler	RL	TFIDF
91.071117	91.022905	82.142177	SMOTE	RL	TFIDF
91.807229	91.335372	83.604316	Random Undersampler	SVC	TFIDF
91.662651	91.231627	83.300351	NearMiss	SVC	TFIDF
98.664131	53.800539	53.254665	OneSidedSelection	SVC	TFIDF
90.613267	90.352048	81.311234	Random Oversampler	SVC	TFIDF
91.122606	91.041352	82.248839	SMOTE	SVC	TFIDF
92.867470	92.959087	86.256126	Random Undersampler	NN	TFIDF
91.903614	91.463415	83.788769	NearMiss	NN	TFIDF
99.069412	71.089588	70.676454	OneSidedSelection	NN	TFIDF
94.404398	94.339653	88.818390	Random Oversampler	NN	TFIDF
94.726002	94.676496	89.456852	SMOTE	NN	TFIDF