information

Article

An Intelligent Approach to Automated Operating Systems Log Analysis for Enhanced Security

Obinna Johnphill [1,*], Ali Safaa Sadiq [1,*], Omprakash Kaiwartya [1] and Mohammad Aljaidi [2]

1 Cyber Security Research Group (CSRG), Department of Computer Science, Nottingham Trent University, Clifton Lane, Nottingham NG11 8NS, UK; omprakash.kaiwartya@ntu.ac.uk
2 Department of Computer Science, Faculty of Information Technology, Zarqa University, Zarqa 13110, Jordan; mjaidi@zu.edu.jo
* Correspondence: obinna.johnphill2022@my.ntu.ac.uk (O.J.); ali.sadiq@ntu.ac.uk (A.S.S.)

Abstract: Self-healing systems have become essential in modern computing for ensuring continuous and secure operations while minimising downtime and maintenance costs. These systems autonomously detect, diagnose, and correct anomalies, with effective self-healing relying on accurate interpretation of system logs generated by operating systems (OSs). Manual analysis of these logs in complex environments is often cumbersome, time-consuming, and error-prone, highlighting the need for automated, reliable log analysis methods. Our research introduces an intelligent methodology for creating self-healing systems for multiple OSs, focusing on log classification using CountVectorizer and the Multinomial Naive Bayes algorithm. This approach involves preprocessing OS logs to ensure quality, converting them into a numerical format with CountVectorizer, and then classifying them using the Naive Bayes algorithm. The system classifies multiple OS logs into distinct categories, identifying errors and warnings. We tested our model on logs from four major OSs; Mac, Android, Linux, and Windows; sourced from Zenodo to simulate real-world scenarios. The model's accuracy, precision, and reliability were evaluated, demonstrating its potential for deployment in practical self-healing systems.

Keywords: multiclass system log classification; operating system log mining; self-healing systems; cybersecurity; CountVectorizer; feature selection; artificial intelligence

check for updates

Citation: Johnphill, O.; Sadiq, A.S.; Kaiwartya, O.; Aljaidi, M. An Intelligent Approach to Automated Operating Systems Log Analysis for Enhanced Security. *Information* **2024**, *15*, 657. https://doi.org/10.3390/info15100657

Academic Editors: Lorenzo Carnevale and Massimo Villari

Received: 3 October 2024
Revised: 10 October 2024
Accepted: 16 October 2024
Published: 19 October 2024

1. Introduction

Self-healing systems are becoming increasingly vital across various domains to ensure continuous operation, minimise downtime, and reduce maintenance costs [1]. These systems utilise a range of techniques to autonomously detect, diagnose, and correct faults [2]. A crucial requirement for self-healing is the ability to identify anomalies or errors within the system. In our context, this involves analysing logs generated by multiple operating systems (OSs) during their operation. However, manually analysing and interpreting large volumes of system logs, especially in complex and dynamic environments, is both tedious and prone to errors [3]. Manual log analysis is typically performed by various IT professionals, including system administrators, security analysts, and network engineers.

- System Administrators: They frequently review system logs to ensure that servers and other infrastructure components are functioning correctly.
- Security Analysts: These professionals analyse logs to identify signs of security breaches or other malicious activities, often as part of a Security Operations Centre (SOC).
- Network Engineers: Network logs are essential for troubleshooting network issues, and network engineers often delve into these logs to diagnose problems.
- DevOps Engineers: In continuous integration and continuous deployment (CI/CD) environments, DevOps engineers review logs to identify issues with deployments and application performance.

- IT Support Teams: When addressing user-reported issues, IT support staff may manually review logs to trace the root cause of a problem.
- Compliance Auditors: They sometimes review logs to ensure that systems comply with various regulations and standards.
- Developers: Developers may perform log analysis during debugging or when analysing the behaviour of their applications in production.

Automated methods for log analysis and classification are necessary to facilitate the self-healing process and circumvent the cumbersome log analysis [4]. Text classification is one such method that can be used to categorise system logs based on their content [5]. The choice of Multinomial Naive Bayes (MNB) and CountVectorizer in this study is grounded in their proven efficacy in text classification tasks, particularly in domains requiring the analysis of large volumes of textual data, such as system logs. MNB is highly regarded for its simplicity and efficiency, making it suitable for scenarios where rapid and reliable classification is essential. It operates effectively on discrete data, which aligns well with the nature of system logs that can be converted into frequency-based features using CountVectorizer. CountVectorizer plays a crucial role in this process by transforming raw textual logs into a structured numerical format, capturing the frequency of terms across the dataset. This step is vital for enabling MNB to process the data effectively. The combination of CountVectorizer and MNB is widely adopted in text classification due to its ability to manage high-dimensional spaces efficiently and its robustness, even when using relatively simple features, as demonstrated in prior research [6,7]. MNB's probabilistic framework allows for the classification of system logs into distinct categories, such as errors or warnings, which is critical for identifying and addressing issues in self-healing systems. By leveraging these methods, our approach ensures that the classification process remains computationally feasible and effective, even when dealing with the large and diverse datasets typical of operating system logs. This combination has been validated in various studies, making it a reliable choice for this research.

In this paper, we present a multiclass text classification approach for self-healing systems using MNB and CountVectorizer on OS logs. The aim is to automatically classify logs into different error and warning categories for Mac, Windows, Android, and Linux platforms. We utilised real system logs that were extracted and preprocessed into CSV files for this purpose. The proposed approach is evaluated using various performance metrics such as accuracy, precision, recall, and F1-score, and is compared with other state-of-the-art methods [8]. The results indicate that our approach achieves high accuracy and recall, demonstrating both efficiency and effectiveness. The proposed approach can be beneficial for various self-healing systems in different domains such as healthcare, transportation, and finance. The proposed methodology demonstrates its effectiveness in detecting errors and warnings in OS logs, providing a reliable foundation for the development of self-healing systems using multiple OS logs. The results of this study showcase the feasibility of leveraging state-of-the-art machine learning techniques to enhance self-healing systems, utilising real-world OS logs [9]. These findings are indicative of the potential for significant improvements in system stability and reliability. Our approach shows great potential for self-healing systems across various domains, marking the beginning of a new era of intelligent and resilient systems.

The increasing complexity of modern computer systems has made them prone to various errors and malfunctions [10]. To address this, self-healing systems have been developed to automatically identify and rectify errors [11]. One of the major challenges in developing self-healing systems is accurately identifying errors and warnings in system logs. These logs can contain vast amounts of data and are often difficult to analyse manually [12]. Therefore, there is a need for automated methods to classify system logs and identify errors and warnings [13]. The problem addressed in this paper is the accurate classification of errors and warnings in OS logs for self-healing systems. The specific challenges include handling the large volume of system logs and accurately classifying a wide range of errors and warnings across multiple operating systems [14]. The primary goal is to develop a text classification model capable of accurately identifying errors and warnings in OS logs, which can then be used to trigger self-healing actions. The primary objective of this research is to develop an effective text classification model for self-healing systems using multiple OS logs from real-world systems. The proposed approach is expected to provide a reliable and efficient solution for the classification of OS logs in self-healing systems, thereby reducing the time required to identify and resolve system issues. The contributions of this study are the following:

1. A novel approach to classify multiple OS logs for self-healing systems using Multinomial Naive Bayes and CountVectorizer.
2. An in-depth analysis of the effectiveness of Multinomial Naive Bayes and CountVectorizer on OS logs from real-world systems.
3. An evaluation of the proposed approach on a dataset consisting of OS logs from Mac, Linux, Windows, and Android systems.

2. Related Work

The rapid evolution of computational systems has led to an increasing demand for innovative methodologies that prioritise mitigating system breakdowns due to anomalies. A pivotal concept in this context is "Computational Self-Healing", which emphasises the autonomous capability of systems to detect, diagnose, and resolve operational inconsistencies. Central to this self-healing paradigm is the proficient classification of logs from various operating systems, enabling proactive anomaly detection and remediation. The application of machine learning techniques, particularly in text-based classification, has proven essential in advancing these systems. Multinomial Naive Bayes (MNB) is a prominent algorithm in this domain. For instance, ref. [15] explored its applicability in classifying and recognising programming language source codes, demonstrating its utility in diverse contexts. The importance of effective feature extraction from textual data cannot be overstated. The study by [16] highlighted the efficacy of CountVectorizer for sentiment analysis in Bengali text, particularly when combined with logistic regression, underscoring its adaptability across different linguistic contexts. Similarly, ref. [17] employed both CountVectorizer and TF-IDF for detecting fake news on Reddit, integrating multiple classifiers, including MNB, to enhance accuracy.

While these studies have established the foundational importance of MNB and CountVectorizer in text classification, there is a notable gap in their application to cross-operating-system log classification, a key requirement for effective self-healing systems. The study further evidences machine learning's adaptability beyond text [18], underscoring its significance in fault detection and system control within integrated energy systems. This adaptability is mirrored in various specialised applications of Naive Bayes, such as [19]'s investigation into hate speech detection in text documents and [20]'s use of Naive Bayes for identifying available website names. These studies illustrate the algorithm's versatility, but they also highlight the need for more targeted approaches in different computational environments. Moreover, previous research has extensively compared various machine learning models. For example, ref. [21] provided a comprehensive evaluation of machine learning and deep learning models, including Support Vector Machines (SVM), K-Nearest Neighbours (KNN), and Convolutional Neural Networks (CNN), for sentiment analysis. Another comparative study by [22] contrasted MNB with a multilayer perceptron for real-time product review classification, showcasing the unique strengths and limitations of each approach.

Despite the breadth of research, the specific challenge of accurately classifying logs across multiple operating systems, critical for self-healing systems, has not been fully addressed. This study seeks to fill this gap by focusing on cross-operating-system log classification, leveraging the strengths of MNB and CountVectorizer. The comparative analysis by [23] between Multinomial and Bernoulli Naive Bayes for text classification further validates the choice of MNB for this research. The study noted that while the Multinomial Naive Bayes classifier considers the frequency of word occurrences within a document, the Bernoulli Naive Bayes classifier operates on the binary concept of whether a term occurs in a document or not. Although both classifiers are used for document classification, they differ significantly in their approach, which has implications for their effectiveness in various contexts.The related work in this area highlights the need for innovative solutions tailored to the unique challenges of cross-operating-system log classification. By building on the established strengths of MNB and CountVectorizer, this research aims to advance the field of self-healing systems, providing a more robust and efficient approach to log analysis across diverse computational environments.

3. Overview of Proposed Method

Our proposed method focuses on classifying multiple OS logs into distinct categories using text classification techniques. Specifically, we employ the CountVectorizer and Multinomial Naive Bayes algorithms for multiclass text classification. The goal is to develop a self-healing system capable of identifying and resolving errors and warnings in real time. To achieve this, we begin by preprocessing the OS logs to extract relevant information, including timestamps, tokens, errors, warnings, and platform types. The preprocessed logs are then transformed into a matrix of token counts using CountVectorizer. This matrix serves as input to the Multinomial Naive Bayes algorithm, which classifies the logs into different categories, such as errors and warnings.

Our solution significantly contributes to the field of self-healing systems by offering a practical approach to real-time error and warning identification and resolution. Our approach is extendable to other types of logs and adaptable to different text classification algorithms. We focus on classifying OS logs based on the type of error or warning in the preprocessed data, as well as the specific OS (Mac, Linux, Windows, and Android). We identify two main categories of issues: errors and warnings.

CountVectorizer converts the textual data into a numerical format, while Multinomial Naive Bayes serves as the classification algorithm. The proposed method aims to enhance self-healing capabilities by automating the identification and classification of anomalies across multiple operating systems, thereby improving system reliability and reducing maintenance costs. Figure 1 illustrates the proposed system, termed the Log Intelligence and Self-Healing System (LISH). LISH comprises two core environments, highlighted in two distinct dotted square boxes in the diagram: the physical environment and the cloud environment. Devices running the operating systems within our study scope, whether situated in the physical or cloud environment, send their system log files via an application programming interface (API) to a virtual log server in the cloud environment. The virtual log server extracts data from the log files, processes them using CountVectorizer, and stores the feature-extracted data in CSV format. The preprocessed data are then transferred to a self-healing virtual server via API, where the Multinomial Naive Bayes algorithm classifies the errors and warnings within the feature-extracted data. Based on predefined thresholds, this classification triggers self-healing remedial actions. Figure 1 provides a comprehensive overview of the self-healing system, illustrating key components and their interactions. The diagram is divided into three main sections, each highlighted with a distinct border style and colour: the green box represents Remedial Actions, the blue-dotted box indicates the Cloud Environment and the orange-dotted box shows the Physical Environment.

3.1. Green Box (Remedial Actions)

The green box, labelled "Remedial Actions", outlines the various actions triggered by the self-healing system in response to detected anomalies. When an anomaly is identified through system log data processing and anomaly detection mechanisms, the system automatically triggers these actions to resolve the issues. The HTTP requests shown in the diagram are part of the communication process between different components, facilitating the transfer of data necessary to initiate these actions. The actions included within this box are the following:

- System Update: Applying updates to software or firmware to keep systems up to date and secure.
- Service Restart: Restarting services that are experiencing issues or have become unresponsive.
- Disk Clean-up: Removing unnecessary files to free up disk space and improve system performance.
- Notification: Alerting administrators or relevant stakeholders about detected issues to enable timely intervention.
- Resource Optimisation: Adjusting system resources to optimise performance and avoid potential bottlenecks.
- Configuration Adjustment: Modifying system configurations to rectify issues or optimise performance settings.
- Load Balancing: Distributing workloads evenly across servers to prevent overload and ensure efficient resource usage.
- Automatic Failover: Automatically switching to a backup system in the event of a failure, ensuring continuous operation.
- Capacity Management: Managing system resources to ensure there is sufficient capacity for current and future operations.

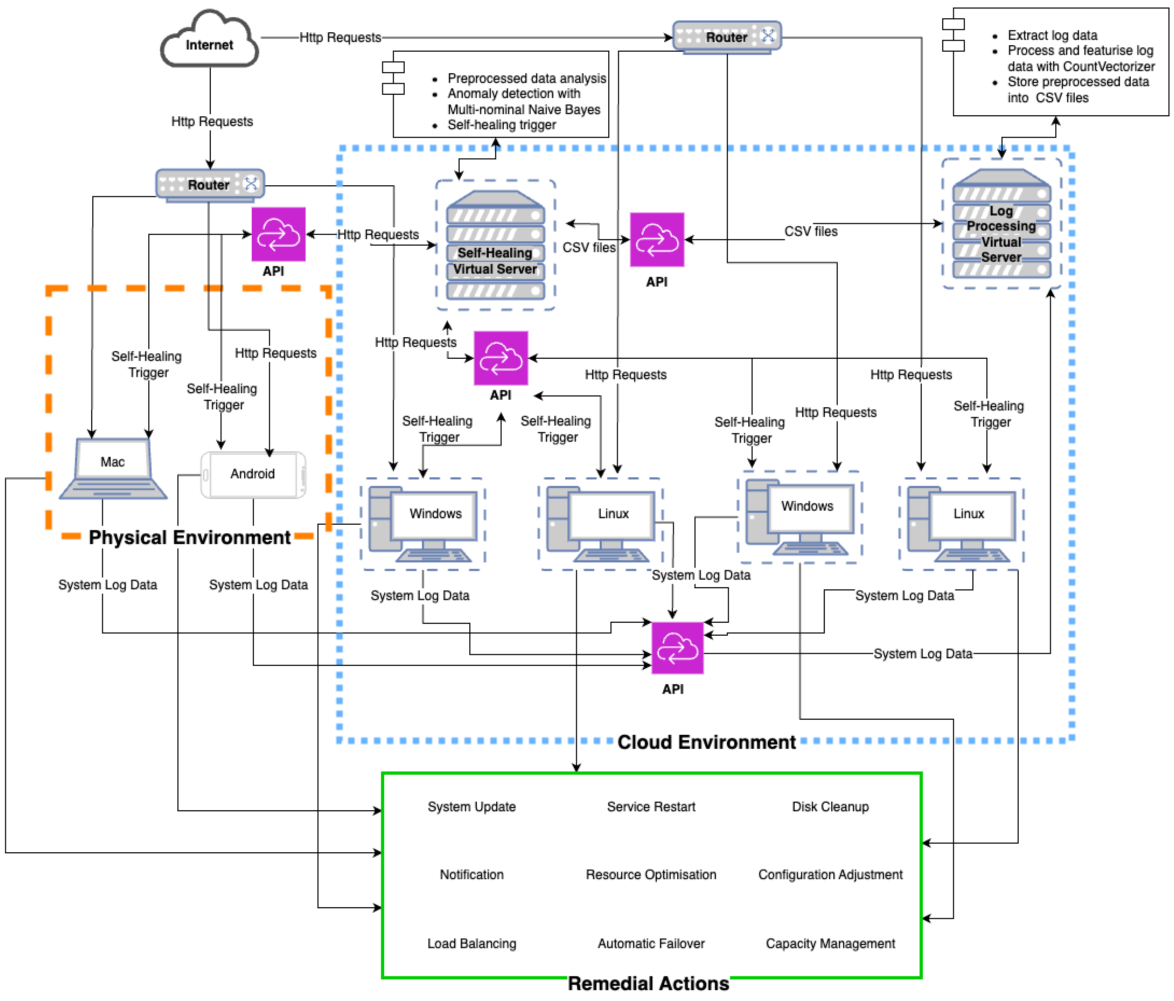

Figure 1. Overview of the proposed system: Log Intelligence and Self-Healing System (LISH).

3.2. Blue-Dotted Box (Cloud Environment)

The blue-dotted box, labelled "Cloud Environment", represents the components hosted within the cloud infrastructure. This environment plays a crucial role in the self-healing process by processing and analysing system logs sent from the physical environment via HTTP requests. The cloud environment consists of the following key components:

- Self-Healing Virtual Server: Handles core processing, including anomaly detection and the triggering of self-healing actions based on the analysed log data.
- Log Processing Virtual Server: Responsible for extracting, processing, and storing log data in a structured format (such as CSV files), preparing them for further analysis.

3.3. Orange-Dotted Box (Physical Environment)

The orange-dotted box, labelled "Physical Environment", represents the physical devices and systems where the operating systems, such as Mac and Android, are running. These devices generate system log data, which are sent to the cloud environment for processing via HTTP requests. The physical environment encompasses the hardware where the operating systems are installed, responsible for generating and transmitting log data to the cloud environment. The self-healing trigger mechanism is initiated when anomalies are detected in these log data, prompting the system to carry out remedial actions to resolve the identified issues.

We developed the machine learning (ML) algorithm that underpins the self-healing trigger mechanism in the system architecture presented in Figure 1. This involves using the Multinomial Naive Bayes (MNB) algorithm with CountVectorizer to classify errors and warnings from system logs. The ML component is integral to identifying anomalies in real time and triggering self-healing actions accordingly. However, the current scope of our work focuses solely on the anomaly detection and classification part of the system. The modules responsible for executing remedial actions, such as system updates, service restarts, and disk clean-ups (outlined in the green box in Figure 1), have not yet been implemented. Developing and integrating these actions into the overall system remains part of our future research efforts. In terms of usage, the self-healing trigger aspect of the system has been conceptualised and developed, but the complete system, including the implementation of remedial actions, has not yet been fully realised. The current system design, as illustrated in Figure 1, shows how the complete solution will function once fully implemented, with the cloud and physical environments interacting through log processing and self-healing virtual servers. We are actively working towards an end-to-end self-healing ML solution. Once the remaining components are developed, we will be able to deploy a comprehensive system that autonomously reacts to fault events, enhances the quality of service (QoS), and reduces costs associated with system failures.

3.4. Data Collection

3.4.1. Description of the Dataset Used

The dataset utilised in our experiment is sourced from Zenodo, specifically from the project titled "Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics". This dataset was published on 1 September 2021 and is available under version v7. It comprises real system error log data extracted from various operating systems, including Windows, Mac, Linux, and Android. These error logs provide valuable insights into system behaviours, errors, and issues encountered during operation. The dataset was curated by LogPAI, a team dedicated to advancing research and applications in log data analysis, particularly in the context of AI-driven log analytics. As the dataset is hosted on Zenodo, an open-access repository, it is openly available for researchers and practitioners to utilise in their studies and applications. It serves as a valuable resource for exploring system behaviours, developing log analytics algorithms, and enhancing system monitoring and troubleshooting techniques.

3.4.2. Sources of the Data

1. Zenodo repository: Project titled "Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics" (Published on 1 September 2021, Version v7)—https://zenodo.org/records/3227177#.ZE5-FuzMJhF (accessed on 20 March 2024).

3.4.3. Data Size and Format

The dataset used in our experiment consists of system error logs collected from multiple operating systems, including Windows, Mac, Linux, and Android. The sizes of the log files for each operating system are as follows:

- Android: 192.3 MB (.log).
- Linux: 2.3 MB (.log).
- Mac: 16.9 MB (.log).
- Windows: 28.01 GB (.log).

The log files are formatted in the standard .log format commonly used for system error logs. The total size of the dataset across all operating systems is 28.22 GB. Each log file contains valuable insights into system behaviours, errors, and issues encountered during operation, making them essential for our research and the analysis of LISH application log data.

3.4.4. Preprocessing Applied to the Data

To prepare the dataset for analysis and modelling, a series of preprocessing steps were applied. These steps ensured that the dataset was cleaned, standardised, and structured in a uniform format, ready for subsequent analysis, modelling, and evaluation. The resulting preprocessed datasets are now well suited for tasks such as log analysis, anomaly detection, and system behaviour prediction. The raw log files obtained from multiple operating systems (Mac, Windows, Android, Linux) underwent the following preprocessing procedures:

Data Cleaning:

- Rows with missing timestamps or tokens were removed from each DataFrame to ensure data integrity and consistency.

Data Type Conversion:

- The "tokens" column in each DataFrame was converted to string type to facilitate further processing.

Missing Value Imputation:

- Missing values in the "error" and "warning" columns were filled using the forward fill method to maintain temporal continuity in the log entries.

Feature Selection:

- Only the necessary columns ("timestamp", "tokens", "error", "warning") were retained in each DataFrame, discarding any extraneous information.

Label Assignment:

- A "Label" column was added to each DataFrame, derived from the filename of the corresponding log files, to facilitate identification and categorisation during analysis.

Data Export:

- The preprocessed data for each operating system was saved into separate CSV files under the "preprocessed-data" directory for ease of access and further experimentation.

3.5. Modelling

3.5.1. Explanation of Multinomial Naive Bayes

The Multinomial Naive Bayes (MNB) classifier is a variant of the Naive Bayes classifier that is particularly well suited to discrete data, especially the type encountered in text classification problems [24–26]. The fundamental "naive" assumption is that the presence (or frequency) of each word is independent of the presence (or frequency) of any other word, given the category. Operating system (OS) logs are records generated by an operating system to track system or application activities. These logs might include information about system events, errors, status updates, or other messages. Classifying these logs is crucial for tasks such as anomaly detection, fault diagnosis, or even understanding the normal operational state of a system. Consider a set of OS log entries where each entry can be categorised into one of several classes, such as "ERROR", "WARNING", "INFO", etc. The Multinomial Naive Bayes approach can be employed to perform this classification. The probabilities described below enable the transformation of raw log data into actionable insights. By classifying OS logs, potential system issues can be quickly identified, or the system's behaviour over time can be better understood.

In the context of operating system (OS) logs, these logs often contain important messages about system health. For example, consider the following log entries:

- Log 1 (ERROR): "Disk failure on drive C".
- Log 2 (WARNING): "Memory usage exceeds 90%".
- Log 3 (INFO): "System backup completed".

The Multinomial Naive Bayes approach simplifies this process by breaking it down into manageable probability calculations, as exemplified in Table 1 and our probability equations, despite operating under the naïve assumption of word independence.

1. Prior Probability:

   The "prior probability" $P(C)$ represents the likelihood that a randomly selected log belongs to a particular class before we look at the content. For example, if 30% of the logs in our dataset are "ERROR" logs, the prior probability of "ERROR" is

$$P(\text{ERROR}) = \frac{\text{Number of “ERROR” logs}}{\text{Total number of logs}} = 0.30$$

2. Likelihood:

   The "likelihood" represents the probability of observing specific words in a log, given that it belongs to a particular class. For example, how likely is it to observe the word "disk" in an "ERROR" log? If the word "disk" appears in 50 out of 300 "ERROR" logs, the likelihood is

$$P(\text{“disk”}|\text{ERROR}) = \frac{50}{300} = 0.167$$

   If the log contains multiple words, such as "disk failure", the overall likelihood is the product of the individual word probabilities, assuming independence:

$$P(\text{“disk failure”}|\text{ERROR}) = P(\text{“disk”}|\text{ERROR}) \times P(\text{“failure”}|\text{ERROR})$$

3. Posterior Probability:

   The "posterior probability" $P(C|w)$ represents the probability that a given log belongs to a specific class, based on the words in the log. Using Bayes' theorem,

$$P(C|w) = \frac{P(w|C)P(C)}{P(w)}$$

   The log entry is classified into the category with the highest posterior probability.

4. Word Probability and Smoothing:

   In cases where certain words are missing from a class during training, Laplace smoothing is applied. This prevents any word from having a probability of zero, which would otherwise cause issues during classification. The probability of a word w_i given a class C is computed as follows:

$$P(w_i|C) = \frac{\text{Count of word } w_i \text{ in logs of class } C + \alpha}{\text{Total number of words in logs of class } C + \alpha d}$$

   where

- α is the smoothing parameter, typically set to 1 to avoid zero probabilities.
- d is the number of unique words across all logs.

5. Example of Classification:

   Suppose we have a new log entry, "disk failure on drive C", and we want to classify it as either "ERROR" or "INFO". The MNB model computes the likelihood of the words appearing in each class and combines it with the prior probabilities. If the posterior probability for "ERROR" is higher than for "INFO", the log is classified as an "ERROR".

6. Processing Logs in Practice:

   In our experiments, the logs are preprocessed and tokenised into their respective words. For each system (e.g., Android, Linux, Windows, Mac), the tokens are evaluated, and summary statistics such as the number of errors, warnings, and the frequency of tokens are calculated. The summary is then used to trigger self-healing actions if thresholds are exceeded. For instance, if the number of "ERROR" logs for Linux exceeds the defined threshold of 100, a self-healing mechanism would be activated to mitigate potential issues:

$$\text{Trigger self-healing if errors} > 100$$

Table 1. Example of Probabilities.

Log Entry	Probability of "ERROR"	Probability of "INFO"
"Disk failure on drive C"	0.85	0.15
"System backup completed"	0.10	0.90

- How Classification Relates to Log Types:
  The classification process involves categorising each log entry into one of the predefined classes (INFO, WARNING, ERROR) based on the content of the logs. By analysing the words within the logs and applying the Naive Bayes algorithm, the system assigns a probability to each class. The class with the highest probability is then selected as the predicted class for that log entry. For example, if a log entry contains multiple instances of the word "error", the system might classify it as an ERROR log with a high probability. Similarly, logs containing terms typically associated with warnings or informational messages would be classified accordingly. The classification is vital for system monitoring and self-healing processes, as it allows the system to prioritise actions based on the severity of the log entries.

3.5.2. Explanation of CountVectorizer

`CountVectorizer` is a popular utility provided by the `scikit-learn` library in Python 3.12, used to convert a collection of text documents into a matrix of token counts. It is essential for transforming text data into a structured, numerical format, which can then be fed into machine learning algorithms. This transformation is crucial because most algorithms require numerical input. The key features of `CountVectorizer` are broken down as follows:

1. Tokenisation: This step involves splitting the string into individual words or terms based on the delimiter, which is whitespace by default.
2. Vocabulary Building: In this step, a vocabulary is constructed using the unique words found in the given corpus of text documents.
3. Encoding: During this step, the occurrences of words from the vocabulary are counted for each document, and each unique word becomes a feature (or column).

When working with text, converting the raw text into a numerical representation is paramount. The Bag-of-Words (BoW) model is one such approach, and `CountVectorizer` essentially implements this model [27]. The BoW model represents text (such as a sentence or document) as an unordered set of words/tokens along with their respective counts, disregarding grammar and word order but retaining word multiplicity.

Relationship between Document, Log Entry, and Log File:

In the context of log analysis, it is important to understand the relationship between a document, log entry, and log file:

- Log File: A log file is a collection of log entries recorded by a system over a period. Each log file typically corresponds to a specific system, application, or service and is stored as a text file on disk.
- Log Entry: A log entry is a single record within a log file. It contains specific details about an event that occurred within the system, such as timestamps, error codes, warnings, or informational messages. Each log entry can be considered analogous to a "document" in text processing.
- Document: In text processing, a "document" refers to any piece of text that we want to analyse or classify. When applying the Bag-of-Words (BoW) model to log analysis, each log entry within a log file is treated as a document. This allows us to convert the log entry into a numerical vector based on the words (tokens) it contains.

  Thus, in this context:

- The log file is a collection of log entries.
- Each log entry is treated as a document in the Bag-of-Words model.

1. Let us consider a corpus D containing m documents (log entries): $D = \{d_1, d_2, \ldots, d_m\}$.
2. The unique words from all documents will form our vocabulary V: $V = \{w_1, w_2, \ldots, w_n\}$.
3. For each document d_i (log entry), its vector representation v_i in the BoW model will be $v_i = [c_1, c_2, \ldots, c_n]$
4. Where c_j represents the count of word w_j in document d_i (log entry).

There are key feature parameters available in `CountVectorizer`, which are briefly described as follows:

1. stop_words: This allows for the specification of "stop words", which are typically common words such as "and", "the", "is", etc. When set, these words are not counted. Some words, often referred to as stop words, are so common that they might not carry significant meaning in certain tasks. Let S be the set of stop words. The modified vocabulary V^1 after removing stop words will be $V^1 = V - S$.
2. ngram_range: This is used to specify the range of n-values for the different n-grams to be extracted. For example, an `ngram_range` of (1,2) would include both individual words and bi-grams (combinations of adjacent two words). Instead of single words, we also consider sequences of words of length k. Such sequences are termed n-grams. For a given `ngram_range` (a, b), the V'' will include n-grams where $a \leq k \leq b$.
3. max_df and min_df: These parameters can be used to set a threshold for words to be included in the vocabulary based on their document frequency. The parameters help to filter out vocabulary based on document frequency. Let $df(w)$ be the number of documents containing the word w. Then, words with $df(w) >$ max_df or $df(w) <$ min_df are excluded from the vocabulary.
4. max_features: This can be used to limit the number of top-occurring words to be considered. It limits the number of features to consider based on their frequency. If set to F, only the top F terms ordered by term frequency across the corpus will be kept.
5. binary: If set to True, it encodes whether a word is present or not, rather than its count.
6. tokeniser: It allows the setting of a custom function or tokeniser for breaking up the text.

3.5.3. Description of the Classification Model and Feature Extraction Method

1. Multinomial Naive Bayes: It is a predictive model which, given input data such as word counts from extraction methods, makes predictions based on the probabilistic relationships it has learned from the training data. It is used for classification tasks and is particularly suitable for features that represent counts, making it apt for text data, especially when processing with techniques such as CountVectorizer.
2. CountVectorizer: It is a processing step that transforms raw text data into a structured, numerical format suitable for machine learning models. It is a feature extraction method whose purpose is to convert a collection of text documents to a matrix of token counts. It tokenises the text documents and builds a vocabulary of known words. For each document, it creates a vector representation of the count of each word or token.

3.5.4. Feature Extraction Process from Raw System Log Files

In this section, we explain the comprehensive process of feature extraction undertaken to harness valuable insights from raw system log files. Our approach involves utilising Python programming along with the Pandas and regular expression libraries to parse, tokenise, and categorise log entries, ultimately generating structured data for analysis and classification. Our feature extraction journey begins with the parsing and tokenisation of log entries. This entails the systematic breakdown of each log entry into its key components for subsequent analysis. The following steps outline our process:

(a) Log File Access: Log files are accessed using the Python "open" function, ensuring correct encoding and handling of potential errors.
(b) Log Entry Segmentation: Each log file is read line by line. Lines without "ERROR" or "WARNING" indications are promptly ignored, focusing solely on relevant entries.
(c) Timestamp Extraction: Regular expressions are employed to extract timestamps from log entries, capturing each event's exact time and date.
(d) Log Message Isolation: The log message itself is obtained by removing any metadata and timestamps, providing a clear and unobstructed text for analysis.
(e) Tokenisation: The log message is then tokenised, breaking it down into individual words. This dissection paves the way for more in-depth content analysis.

During this feature extraction process, we assign severity levels to each log entry based on the content of the log messages. The following key steps elucidate this process:

(a) Timestamp: The timestamp extracted from each log entry, reflects the precise occurrence time.
(b) Tokens: The tokenised version of the log message facilitates granular content analysis.
(c) Severity Flags: Binary flags indicating the presence of certain severity levels, like "error" and "warning", within the log message. These flags hold significant importance for later classification tasks.

4. Challenges and Limitations Encountered During Preprocessing and Model Analysis

4.1. Preprocessing Challenges

During our experiment, we encountered several challenges related to the preprocessing of log data, particularly with Linux logs. Unlike the logs from Mac, Android, and Windows, which exhibited a consistent structure and timestamp format, the Linux logs presented a unique challenge due to their different timestamp format. This discrepancy necessitated the creation of custom code specifically designed to handle Linux logs. The original preprocessing pipeline, which worked seamlessly for the other operating systems, failed to process Linux logs correctly due to this format difference. To address this issue, we developed a customised regular expression (regex) formula tailored to the Linux timestamp format. This specialised approach allowed us to accurately parse and extract the necessary data from Linux logs, ensuring that they could be processed and analysed alongside logs from other systems. Once the logs were parsed, we applied the CountVectorizer algorithm for feature extraction and tokenisation, preparing the data for machine learning classification. The processed data were then organised into a structured format using the Pandas library and saved in CSV format to facilitate subsequent analysis.

4.2. Memory Issues and Model Analysis

Another significant challenge encountered during our study was related to memory limitations, particularly when processing a large volume of Windows log data. The size of this dataset exceeded the available memory capacity, preventing us from combining the model analysis into a single processing block of code. This limitation required us to separate the dataset by the operating system, thereby reducing the load on memory during processing. The separation of datasets allowed us to perform model analysis on each operating system independently, but it also introduced additional complexity in managing and coordinating the analyses across different systems. This approach, while necessary given our memory constraints, is not ideal for achieving a fully integrated analysis and self-healing trigger across all systems simultaneously. The future availability of larger memory resources or access to a supercomputer would allow us to overcome this limitation. With enhanced computational resources, we could potentially combine our algorithm into one processing block and run the analysis and self-healing mechanisms for all systems in a unified, simultaneous process. This advancement would significantly streamline the analysis workflow and enhance the efficiency and scalability of the self-healing system.

- Log Entry Retrieval: Log entries are retrieved from the multiple OS log files using regular expressions. Regular expressions (regex) are utilised to search for and extract specific patterns within the log files. For instance, a regex pattern might be employed to identify the start of a new log entry, typically indicated by a timestamp or a particular log level (e.g., “INFO”, “ERROR”, “WARNING”). These patterns assist in isolating individual log entries from the continuous stream of text in a log file. Commenting on the regular expressions would involve explaining the purpose of each pattern, such as matching timestamps in different formats, extracting log levels, or identifying key phrases indicative of specific events.
- Individual Entry Processing: Each log entry undergoes processing to extract relevant information, including the timestamp, tokens (the individual words or terms), and severity flags (such as “ERROR” or “WARNING”). Regular expressions might also be used in this step to precisely extract timestamps and other critical data. For example, a regex could be designed to capture timestamps in formats like “YYYY-MM-DD HH:MM:SS” or to isolate specific tokens that indicate an error or warning. Commenting on these regular expressions would clarify how they are constructed to match various timestamp formats or extract specific tokens.
- Enrichment with “ALERT” Warnings: An additional step involves identifying “ALERT” tokens within the log entries and flagging them accordingly for warning indications. Regular expressions are likely employed here to search for occurrences of the term “ALERT” (or similar terms) within the log entries. If an “ALERT” is detected, the log entry is flagged to indicate a higher severity level. Comments on the regular expressions used in this context would describe how the patterns are designed to detect variations of the word “ALERT” and differentiate between different levels of severity.

Our feature extraction process transforms raw log data into a structured and enriched format. These structured data serve as the foundational input for subsequent machine learning classification tasks, and a summary of the extracted data saved into CSV files is highlighted in Table 2. By systematically parsing, tokenising, and categorising log entries, we enabled accurate and automated identification of error and warning levels within system log files. The logs considered span various time frames depending on the operating system and the specific system’s logging practices. Subsequent sections will delve into the utilisation of machine learning techniques to leverage these derived features for enhanced log analysis and classification.

Table 2. Summary of extracted log entries.

File Name	Number of Entries
Linux_extracted.csv	25,567
Mac_extracted.csv	107,201
Windows_extracted.csv	114,422,305
Android_extracted.csv	1,555,005

4.2.1. Analysis and Self-Healing Trigger Based on Extracted Log Data

In this section, we present the results of our feature extraction process applied to the preprocessed log files obtained from various operating systems. We delve into the insights derived from the extracted features and discuss potential self-healing triggers based on the error and warning thresholds. Moreover, visualisations are employed to enhance the understanding of the data distribution and enable system-specific comparisons. The logs analysed in this study cover different time spans depending on the source systems, and the workflow was designed to ensure a consistent approach to preprocessing, as summarised below. Table 3 provides an overview of the feature-extracted data statistics for each system, including the number of errors and warnings, the average number of tokens, and the sum of labels for each system. Analysis of the feature-extracted data provides insights into the error and warning distribution across different systems. The self-healing triggers help identify potential areas that require attention for system stability. The visualisation expands the understanding of data patterns, fostering a deeper understanding of the log entry characteristics. These insights serve as a foundation for subsequent actions and strategies aimed at enhancing system reliability and performance.

Table 3. Feature extracted data summary.

System	Error	Warning	Tokens	Label
Android	1431.0	3.0	9.357741	1.03×10^6
Linux	146.0	250.0	9.048304	3.27×10^8
Mac	356.0	381.0	11.383989	2.71×10^5
Windows	142,113.0	0.0	6.015923	1.01×10^{10}

4.2.2. Explanation of Our Log Preprocessing Workflow

Figure 2 represents the overall process of log extraction, preprocessing, feature extraction, model classification, and self-healing classification. The system processes log files from different operating systems (Mac, Windows, Android, and Linux) and ensures that the logs are processed, analysed, and actions are triggered when necessary.

Workflow Steps:

The system follows these steps:

- Log Extraction: The system starts by ingesting logs from various systems.
- Log Parsing: This step extracts relevant data (such as timestamps, errors, and warnings) from the logs.
- Data Cleansing: The logs are cleaned to remove incomplete or irrelevant data.
- Feature Extraction: Key features such as error counts and log tokens are identified for further processing.
- Model Classification: The data are classified using a machine learning model to predict the occurrence of issues or events.
- Self-Healing Classification: Based on thresholds for errors and warnings, the system decides whether a self-healing action is required.
- Visualisation and Reporting: The system generates reports and visualisations to summarise the health of the monitored systems.

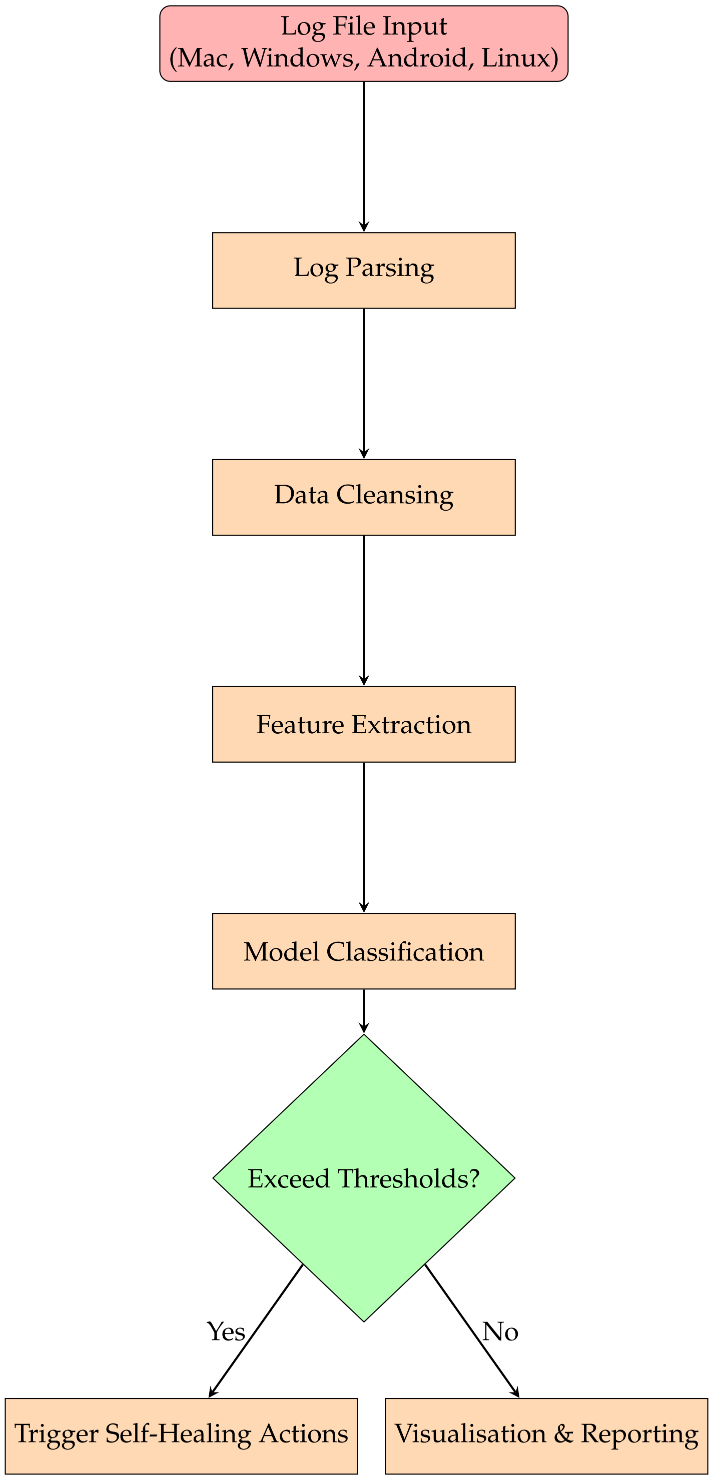

Figure 2. Flow diagram for log extraction, preprocessing, feature extraction, model classification, and self-healing monitoring.

We developed a custom codebase to perform preprocessing on log data from multiple systems (Mac, Windows, Android, and Linux). The logs were collected over varying periods depending on the system, reflecting different operational environments. As a result of this preprocessing process, each system's log data were cleaned, missing values were addressed, unnecessary columns were removed, and the data were structured for further analysis or machine learning tasks. The preprocessed data were saved in separate CSV files for each system, with an added "Label" column to identify the source of the logs. Below is a step-by-step breakdown of the preprocessing process we implemented:

1. Reading the Log Files: We read log files for each system (`Mac_extracted.csv`, `Windows_extracted.csv`, `Android_extracted.csv`, and `Linux_extracted.csv`) using the Pandas library's `pd.read_csv()` function. Each CSV file contains logs with various columns like timestamp, tokens, error, and warning.
2. Dropping Rows with Missing Values: For each system's DataFrame (`df_mac`, `df_win`, `df_android`, `df_linux`), we dropped rows with missing values in the `timestamp` or `tokens` columns using the `dropna()` function. This ensures that only complete records are used in the analysis.
3. Converting Tokens to String Type: For each system's DataFrame, we converted the `tokens` column to a string type using the `astype()` function, ensuring a consistent data type for subsequent processing steps.
4. Filling Missing Error and Warning Values: We filled missing values in the `error` and `warning` columns using forward fill (`ffill`). This ensures that missing values are replaced with the most recent non-missing value, maintaining temporal continuity in the data.
5. Extracting Relevant Columns: For each system's DataFrame, we extracted only the relevant columns (`timestamp`, `tokens`, `error`, `warning`) using indexing. This step focuses the dataset on the most important features for further analysis.
6. Adding a Label Column: We added a `Label` column to each system's DataFrame. The label is derived from the filename by splitting the filename to extract the required portion, aiding in the identification and categorisation of the logs during analysis.
7. Saving Preprocessed Data: The preprocessed data for each system are saved to separate CSV files using the `to_csv()` function. The index is set to `False` to avoid saving the default index column. The resulting CSV files generated are as follows:

5. Threshold Determination and Impact on System Performance

5.1. Thresholds Determination

In our research, the thresholds for triggering self-healing actions were meticulously determined through an analysis of log data from multiple operating systems. These thresholds play a crucial role in ensuring that the self-healing mechanism activates appropriately, balancing the need for responsive intervention with the avoidance of unnecessary actions that could degrade system performance. The process of setting these thresholds involved a thorough examination of the frequency and severity of errors and warnings within the log files. Specifically, we established an error threshold of 100 and a warning threshold of 500. These values were chosen based on the average frequency of occurrences in the dataset, ensuring that the system triggers self-healing actions only when a significant number of errors or warnings are detected, indicating a potential issue that requires intervention.

5.2. Impact on System Performance

The chosen thresholds have a direct impact on the system's performance. Setting these thresholds too low could result in frequent, unnecessary self-healing actions, potentially leading to performance degradation due to the system being overly cautious. Conversely, setting them too high could delay necessary interventions, allowing errors or warnings to escalate into more significant issues. By analysing the frequency and distribution of errors and warnings across different operating systems, we ensured that the thresholds were optimised for each environment. This optimisation process involved balancing the need for prompt corrective action to maintain overall system efficiency. For future implementation, as system logs grow in complexity and volume, these thresholds may need to be revisited and adjusted. Access to more substantial computational resources, such as supercomputers, would allow for real-time dynamic threshold adjustments, further enhancing the self-healing system's adaptability and responsiveness.

5.2.1. Analysis of the Extracted and Preprocessed Data

We performed analysis on the extracted and preprocessed log data from different systems (Android, Linux, Mac, Windows) and visualised the number of errors and warnings for each system using bar charts. Here is a breakdown of the analysis and the resulting visualisations:

1. Data Loading and Preprocessing:
   - (a) The code starts by reading the extracted log files for each system and concatenates them into a single dataframe.
   - (b) Rows with missing or incomplete timestamp or tokens information are dropped, and the "tokens" column is converted to a string type.
   - (c) Missing error and warning values are filled using forward fill, ensuring that missing values are replaced with the most recent non-null value.
   - (d) The dataframe is then narrowed down to include only the relevant columns ("timestamp", "tokens", "error", "warning").
2. Saving Preprocessed Data:
   - (a) The preprocessed data for each system are saved into separate CSV files in the specified directory.
3. Bar Chart Creation (Extracted Data):
   - (a) The code calculates the number of errors and warnings for each system in the extracted data and stores them in the `num_errors_extracted` and `num_warnings_extracted` lists.
   - (b) A bar chart is created using Matplotlib, displaying the number of errors and warnings for each system in the extracted data. Errors and warnings are represented with stacked bars for each system.
4. Bar Chart Creation (Preprocessed Data):
   - (a) The preprocessed data for each system are read from the saved CSV files.
   - (b) The code calculates the total number of errors and warnings for each system in the preprocessed data and stores the results in the `num_errors_preprocessed` and `num_warnings_preprocessed` lists.
   - (c) Another bar chart is created to visualise the number of errors and warnings for each system in the preprocessed data, with stacked bars for errors and warnings.
5. Displaying Plots:
   - (a) Both bar charts are displayed using plt.show().

5.2.2. Analysis of Extracted and Preprocessed Data Counts

Table 4 provides a comprehensive overview of the extracted and preprocessed data counts for errors and warnings across different operating systems. The table is divided into two main sections:

- Extracted Data Error Count and Warning Count:
  - Lists the total number of errors and warnings identified during the initial extraction of log data from each operating system.
  - The datasets analysed include Android, Linux, Mac, and Windows.
  - The initial extracted logs contain raw, unstructured data directly sourced from system log files, which often include redundant, irrelevant, or incomplete entries.
- Preprocessed Data Error Count and Warning Count:
  - Shows the error and warning counts after preprocessing, which involves cleaning and structuring the data. The reduction in the number of errors and warnings in the preprocessed data reflects the removal of irrelevant entries and consolidation of the log entries, ensuring that the dataset is more focused and ready for subsequent analysis. Preprocessing steps include:

Removal of irrelevant entries: Log entries that do not provide meaningful information for error or warning detection, such as routine informational logs (e.g., "System initialisation completed"), are removed. These logs are often verbose but do not contribute to the identification of system faults.

Handling missing data: Missing values in crucial fields such as "timestamp" and "tokens" are dropped. This ensures that only complete entries, which provide sufficient context for the analysis, are retained. Incomplete records, which may hinder the accuracy of predictions, are eliminated.

Forward-filling: Missing values in the "error" and "warning" columns are filled using a forward-fill method. This assumes that the last known error or warning persists until it is explicitly resolved in the logs. This technique is particularly useful when logs do not consistently document the resolution of an issue.

Consolidation of repetitive log entries: Frequently repeated logs, such as those generated by regular status updates or background processes, are consolidated. This reduces redundancy, focusing the dataset on significant events rather than routine, non-critical messages.

Tokenisation and standardisation of format: The "tokens" column, which holds the main log message, is processed by converting it to a consistent format (e.g., standardising case or removing special characters) to prepare the data for further analysis and classification.

Table 4. Extracted and preprocessed data counts.

Extracted Data Error Count		**Extracted Data Warning Count**	
Dataset	**Error Count**	**Dataset**	**Warning Count**
Android	1.555×10^6	Android	1.555×10^6
Linux	25,567	Linux	25,567
Mac	107,201	Mac	107,201
Windows	1.14422×10^8	Windows	1.14422×10^8
Preprocessed Data Error Count		**Preprocessed Data Warning Count**	
Dataset	**Error Count**	**Dataset**	**Warning Count**
Android	154,232	Android	3237
Linux	415	Linux	4198
Mac	16,661	Mac	370
Windows	329,807	Windows	202,860

We assess potential self-healing triggers based on the error and warning thresholds using the data shown in Figure 3 (extracted error and warning counts) and Figure 4 (preprocessed error and warning counts). Table 4 summarises the extracted and preprocessed data counts for each dataset. Figure 4 shows preprocessed data error count for Android: 154,232, Linux: 415, Mac: 16,661, and Windows: 329,807.

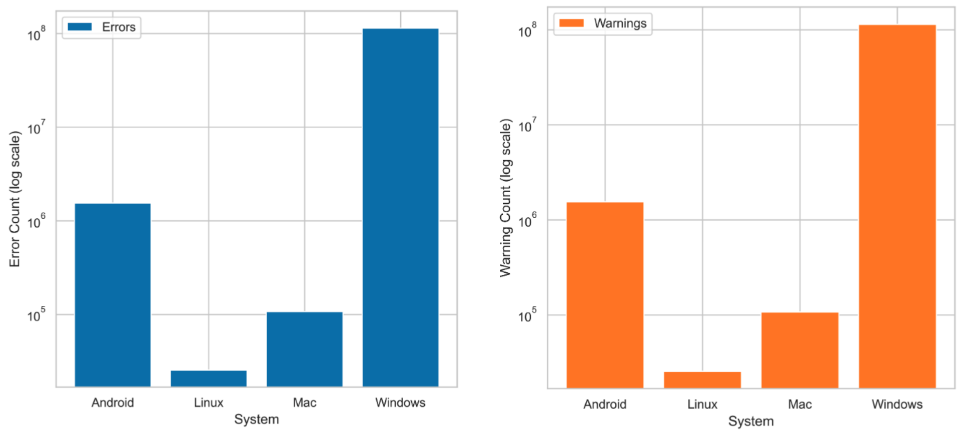

Figure 3. Extracted data counts: error count (**left**) and warning count (**right**).

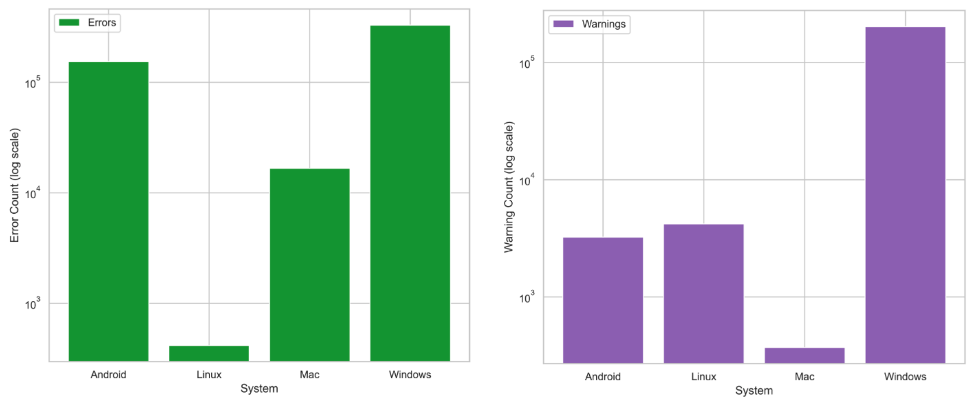

Figure 4. Preprocessed data counts: error count (**left**) and warning count (**right**).

5.2.3. Analysis of the Extracted and Preprocessed Data

Table 5 provides a detailed overview of the error statistics across different operating systems: Android, Linux, Mac, and Windows. The statistics include the count, mean, standard deviation (std), minimum (min), 25th percentile (25%), 50th percentile (50%), 75th percentile (75%), and maximum (max) values for errors recorded in the logs of each system.

- Count: The count represents the total number of log entries considered in the analysis for each operating system. For instance, in the Windows system, over 114 million log entries were analysed.
- Mean: The mean value is the average number of errors observed per log entry across all log entries for each operating system. For example, the mean error rate for Android logs is approximately 0.0992, indicating that errors are relatively less frequent.
- Standard Deviation (std): This value measures the variability or spread of the error counts around the mean. A higher standard deviation indicates greater variability in the error rates across log entries. For example, Mac has a standard deviation of approximately 0.362, suggesting more variability in error occurrences compared to the other systems.
- Minimum and Maximum Values: These indicate the range of error counts observed across log entries. The minimum value is 0 for all systems, showing that there are log entries without any errors, while the maximum is 1, indicating that errors are binary (either present or absent).

Table 5. Error statistics across systems.

Statistic	Android	Linux	Mac	Windows
count	1.555005×10^6	25,567.000000	107,201.000000	1.144223×10^8
mean	9.918425×10^{-2}	0.016232	0.155418	2.882366×10^{-3}
std	2.989093×10^{-1}	0.126369	0.362305	5.361024×10^{-1}
min	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	0.000000
50%	0.000000	0.000000	0.000000	0.000000
75%	0.000000	0.000000	0.000000	0.000000
max	1.000000	1.000000	1.000000	1.000000

Table 6 provides a detailed overview of the warning statistics across different operating systems: Android, Linux, Mac, and Windows. The statistics include the count, mean, standard deviation (std), minimum (min), 25th percentile (25%), 50th percentile (50%), 75th percentile (75%), and maximum (max) values for warnings recorded in the logs of each system.

- Count: Similar to Table 5, the count reflects the total number of log entries analysed for warnings in each operating system.

- Mean: The mean value represents the average frequency of warnings per log entry. For instance, the mean warning rate for Linux logs is 0.164, indicating that warnings are more frequently logged in Linux compared to other systems.
- Standard Deviation (std): This measures the dispersion of warning counts around the mean. A higher standard deviation, as seen in Linux (0.370), suggests that the warning frequencies vary significantly across different log entries.
- Minimum and Maximum Values: These values highlight the range of warnings across log entries. The minimum value of 0 indicates the presence of entries without any warnings, while the maximum value of 1 shows that warnings, like errors, are binary in nature.

Table 6. Warning statistics across systems.

Statistic	Android	Linux	Mac	Windows
count	1.555005×10^6	25,567.000000	107,201.000000	1.144223×10^8
mean	2.081665×10^{-3}	0.164196	0.003451	1.772906×10^{-3}
std	4.557777×10^{-2}	0.370461	0.058648	4.206855×10^{-2}
min	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	0.000000
50%	0.000000	0.000000	0.000000	0.000000
75%	0.000000	0.000000	0.000000	0.000000
max	1.000000	1.000000	1.000000	1.000000

The numbers in the tables Tables 4–6 are presented in scientific notation, such as 1.14422×10^8, which means 1.14422×10^8, and 1.555×10^6, which means 1.555×10^6.

The mean and standard deviation values were calculated based on the total number of log entries (count) in each system. These statistics provide insights into the distribution and variability of errors and warnings across different operating systems, thereby facilitating a comparative analysis of system reliability and alert frequencies.

The triggers identified for each system, as shown in Table 7, are highlighted as follows:

1. For the Android system, neither errors nor warnings exceeded the thresholds. No self-healing action is triggered.
2. For the Linux system, both errors and warnings exceeded the thresholds. A self-healing action is triggered to address the high error and warning counts.
3. For the Mac system, errors exceeded the threshold, but warnings did not. A self-healing action is triggered for the Mac system to address the high error count.
4. For the Windows system, warnings exceeded the threshold, but errors did not. A self-healing action is triggered for the Windows system to address the high warning count.

Table 7. Errors and warnings with thresholds.

Operating System	Errors	Warnings
Android	0.0	0.0
Linux	1.0	1.0
Mac	1.0	0.0
Windows	0.0	1.0

5.2.4. Libraries and Software Used

Our analysis was conducted using a comprehensive suite of Python libraries, integrated through the Anaconda distribution and executed within Jupyter Notebooks. The primary tools and libraries employed in our research include the following:

- Pandas: A powerful data manipulation and analysis library for Python, used for handling and preprocessing the dataset [28].

- NumPy: Essential for numerical computations, providing support for arrays and mathematical functions [29].
- Matplotlib and Seaborn: Both libraries were utilised for creating static, interactive, and informative plots to visualise the data [30,31].
- Scikit-learn: A machine learning library used for implementing the Multinomial Naive Bayes, Logistic Regression, Linear Discriminant Analysis, and Gradient Boosting classifiers, as well as for data preprocessing and evaluating model accuracy [32].
- PyCaret: An open-source, low-code machine learning library that significantly expedited the model selection process by automating the comparison of different algorithms [33].
- Tabulate: Facilitated the presentation of results in tabular format, enhancing the readability of our findings [34].

5.2.5. Hardware Specification

The computational analyses were performed on a MacBook Pro with the following hardware specifications:

- Processor: Apple M3 Pro.
- Memory: 18 GB.
- Operating System: macOS 14.5 (23F79).

The current hardware setup, featuring the Apple M3 Pro chip, 18 GB of memory, and macOS 14.5 (23F79), completed the data extraction in 34 min. However, we believe that this computation is slow due to the vast volume of log data we are processing. Despite the robust capabilities of this hardware configuration, the sheer size of the dataset imposes significant demands on processing time.

To enhance data management efficiency, we implemented the following optimisations:

- Chunk-Based Processing: The `parse_log_file_in_chunks` function processes the log file in manageable chunks (e.g., 10,000 lines), allowing for efficient handling and processing of large datasets.
- Incremental Data Processing: Processed chunks are saved incrementally to the output CSV file, mitigating memory overload.
- Proactive Garbage Collection: Explicit calls to `gc.collect()` are made to reclaim memory.

Despite these optimisations, our analysis indicates that a significantly higher RAM capacity server will be essential to meet the computational demands of our proposed LISH system for real-time log data extraction and preprocessing.

5.3. Model Selection and Training Using Multinomial Naive Bayes

The foundation of our study is the utilisation of real log data for Android, Mac, Linux, and Windows systems, sourced from the Zenodo platform. This comprehensive dataset includes a broad range of system logs, providing an invaluable resource for the effective analysis and classification of system errors and warnings. Given the categorical nature of log messages and our objective to classify these messages into predefined categories such as "error" or "warning", we selected the Multinomial Naive Bayes (MNB) model. The MNB algorithm is particularly well suited for text classification tasks due to its efficiency in handling categorical data, making it an ideal choice for this study. To enhance the robustness and applicability of our model, we began by working with the extracted and preprocessed datasets. The preprocessing phase involved cleaning, tokenising, and structuring the log data, followed by feature extraction using CountVectorizer, which converted the textual log data into a matrix of token counts. This matrix served as the input for the MNB classifier. To evaluate the performance of the MNB model in comparison to other machine learning models, we utilised PyCaret, an open-source, low-code machine learning library. PyCaret allowed us to systematically train and assess multiple models, including MNB, on the preprocessed datasets. By employing PyCaret's built-in tools, we were able to compare the performance of MNB with other classifiers such as Logistic Regression, Random Forest, and

Support Vector Machines, among others. Performance metrics such as accuracy, precision, recall, and F1-score were used to assess the effectiveness of each model. Notably, the MNB model demonstrated a particularly high recall rate, outperforming other models in identifying true positives, which is crucial for accurately detecting errors and warnings within the logs. This high recall performance is a significant advantage, especially in scenarios where it is more critical to identify all potential issues (even at the expense of a few false positives) than to achieve the highest precision. The results indicated that while the MNB model was exceptionally efficient in terms of recall, it could be complemented by other models to achieve potential gains in precision and overall classification performance. This comparative analysis highlighted the importance of selecting appropriate models based on the specific characteristics of the dataset and the classification task at hand. The insights gained from this evaluation underscore the value of combining different machine learning models to enhance the accuracy and reliability of system log classification across varied operating systems.

5.3.1. Data Preprocessing

Prior to model training, the dataset underwent a rigorous preprocessing phase. This phase involved reading and cleaning log files from each system (Android, Mac, Linux, and Windows), followed by a tokenisation process to convert log messages into a format suitable for analysis. Specifically, preprocessing steps included the following:

1. Data Extraction: Importing log data from CSV files, each representing a distinct system.
2. Cleaning: Dropping rows with missing values and converting the “tokens" column to a string type to ensure data consistency.
3. Feature Engineering: Calculating summary statistics such as the total number of errors and warnings, and the average number of tokens per log entry.

5.3.2. Model Training

With the preprocessed data, we proceeded to train the Multinomial Naive Bayes model. The choice of MNB was motivated by its suitability for the text classification of the log messages, aiming to classify them based on their severity (error or warning) and type (system-specific issues). The training process entailed the following steps:

1. Vectorisation: Applying TF-IDF vectorisation to convert the tokenised log messages into numerical features, enabling the MNB algorithm to process and learn from the text data.
2. Model Fitting: Utilising the sci-kit-learn library’s `Multinomial Naive Bayes` class to fit the model on the training dataset, which comprises a subset of the preprocessed log data.
3. Hyperparameter Tuning: Although MNB requires minimal hyperparameter tuning, we explored different configurations of the model’s parameters to optimise performance.

6. Model Performance Analysis Results

We conducted a performance analysis of the Multinomial Naive Bayes (MNB) algorithm in comparison to other classification models. Our results indicate the viability of MNB as a tool for implementing self-healing functionality in cyber-physical systems. MNB excelled in recall across all metrics, demonstrating its effectiveness in identifying errors and warnings. Although logistic regression outperformed MNB in accuracy, MNB still achieved commendable accuracy scores. The performance of multiple classifiers was evaluated using several metrics: accuracy, AUC (area under the curve), recall, precision, F1-score, kappa, and MCC (Matthews correlation coefficient), along with their training time (TT) in seconds. The following tables summarize the results:

1. Android Dataset Results

Table 8 summarises the performance of different classifiers on the Android dataset. The Multinomial Naive Bayes (MNB) classifier achieved an accuracy of 98.87% and a recall of 56.21%, which is particularly notable for its ability to correctly identify positive instances.

The precision and F1-scores were also competitive, showing a balanced performance. Importantly, MNB had one of the lowest training times, making it suitable for rapid updates.

Table 8. Model performance analysis on Android dataset.

Model	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC	TT (s)
knn	0.9604	0.0000	0.6007	1.0000	0.7506	0.7305	0.7586	7.1660
dt	0.9604	0.0000	0.6007	1.0000	0.7506	0.7305	0.7586	1.1400
svm	0.9604	0.9903	0.6007	1.0000	0.7506	0.7305	0.7586	1.2020
rf	0.9604	0.0000	0.6007	1.0000	0.7506	0.7305	0.7586	2.5570
ada	0.9604	0.8004	0.6007	1.0000	0.7506	0.7305	0.7586	1.1710
gbc	0.9604	0.8262	0.6007	1.0000	0.7506	0.7305	0.7586	3.3610
et	0.9604	0.0000	0.6007	1.0000	0.7506	0.7305	0.7586	1.1460
lightgbm	0.9604	0.0000	0.6007	1.0000	0.7506	0.7305	0.7586	1.1710
lr	0.9603	0.9991	0.6010	0.9987	0.7504	0.7303	0.7582	1.3720
lda	0.9562	0.9847	0.5634	0.9918	0.7186	0.6960	0.7297	1.1680
ridge	0.9559	0.9847	0.5588	0.9936	0.7153	0.6934	0.7273	1.1380
qda	0.9008	0.0000	0.4892	0.9000	0.6118	0.6960	0.7297	1.2390
dummy	0.9008	0.0000	0.4892	0.9000	0.6118	0.6960	0.7297	1.1710
nb	0.1013	0.0000	1.0000	0.0994	0.1808	0.0005	0.0152	1.1190

2. Linux Dataset Results

Table 9 presents the performance metrics for the Linux dataset. MNB, again, showed robust performance with high accuracy and recall rates. The classifier's training time was minimal compared to other models, reinforcing its efficiency and applicability in real-time systems.

Table 9. Model performance analysis on Linux dataset.

Model	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC	TT (s)
nb	0.9887	0.0000	0.5621	0.7041	0.6173	0.6117	0.6196	0.0190
knn	0.9877	0.0000	0.2414	1.0000	0.3847	0.3809	0.4839	0.0910
dt	0.9875	0.0000	0.2310	1.0000	0.3703	0.3666	0.4722	0.0210
svm	0.9875	0.9107	0.2310	1.0000	0.3703	0.3666	0.4722	0.0510
ridge	0.9875	0.9104	0.2310	1.0000	0.3703	0.3666	0.4722	0.0190
rf	0.9875	0.0000	0.2310	1.0000	0.3703	0.3666	0.4722	0.0370
ada	0.9875	0.6155	0.2310	1.0000	0.3703	0.3666	0.4722	0.0330
gbc	0.9875	0.6155	0.2310	1.0000	0.3703	0.3666	0.4722	0.0390
lda	0.9875	0.8940	0.2310	1.0000	0.3703	0.3666	0.4722	0.0220
et	0.9875	0.0000	0.2310	1.0000	0.3703	0.3666	0.4722	0.0330
lightgbm	0.9875	0.0000	0.2310	1.0000	0.3703	0.3666	0.4722	0.0260
lr	0.9849	0.9035	0.0690	0.9000	0.1270	0.1253	0.2427	0.1590
qda	0.9838	0.0000	0.0690	0.9000	0.1270	0.1253	0.2427	0.0190
dummy	0.9838	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0180

3. Mac Dataset Results

As seen in Table 10, the Mac dataset results highlight the MNB classifier's consistency. The model maintained high accuracy and a competitive recall rate. The precision and F1-scores were stable, indicating a reliable performance across different metrics. The quick training time further supports the use of MNB in dynamic environments.

Table 10. Model performance analysis on Mac dataset.

Model	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC	TT (s)
lr	0.9400	0.9714	0.6147	0.9993	0.7611	0.7291	0.7573	0.1980
knn	0.9398	0.0000	0.6126	0.9999	0.7597	0.7275	0.7561	0.1410
et	0.9216	0.0000	0.4956	0.9999	0.6549	0.6177	0.6697	0.1210
lda	0.9211	0.9626	0.4975	0.9957	0.6622	0.6230	0.6705	0.0620
ridge	0.9195	0.9716	0.4876	0.9904	0.6611	0.6202	0.6705	0.0620
dt	0.9033	0.0000	0.3775	1.0000	0.5480	0.5060	0.5819	0.0630
svm	0.9033	0.9731	0.3775	1.0000	0.5480	0.5060	0.5819	0.0630
rf	0.9033	0.0000	0.3775	1.0000	0.5480	0.5060	0.5819	0.0630
ada	0.9033	0.6888	0.3775	1.0000	0.5480	0.5060	0.5819	0.0660
gbc	0.9033	0.7341	0.3775	1.0000	0.5480	0.5060	0.5819	0.1590
lightgbm	0.9033	0.0000	0.3775	1.0000	0.5480	0.5060	0.5819	0.1560
qda	0.8446	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0620
dummy	0.8446	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0610
nb	0.1588	0.0000	1.0000	0.1560	0.2698	0.0012	0.0249	0.0610

4. Windows Dataset Results

The Windows dataset performance metrics are displayed in Table 11. MNB excelled with a high accuracy rate and significantly lower training time than other classifiers. The model's ability to maintain stability in precision and F1-scores across diverse datasets makes it an ideal candidate for self-healing systems that require dependable and efficient models.

Table 11. Model performance analysis on Windows dataset.

Model	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC	TT (s)
knn	0.9999	0.0000	0.9732	1.0000	0.9864	0.9864	0.9865	24.9233
dt	0.9999	0.0000	0.9732	1.0000	0.9864	0.9864	0.9865	3.5067
rf	0.9999	0.0000	0.9732	1.0000	0.9864	0.9864	0.9865	8.2633
ada	0.9999	0.9866	0.9732	1.0000	0.9864	0.9864	0.9865	4.3900
gbc	0.9999	0.9793	0.9732	1.0000	0.9864	0.9864	0.9865	34.6100
et	0.9999	0.0000	0.9732	1.0000	0.9864	0.9864	0.9865	5.7700
lightgbm	0.9999	0.0000	0.9732	1.0000	0.9864	0.9864	0.9865	4.8600
lr	0.9998	0.9993	0.9305	0.9938	0.9639	0.9634	0.9661	3.9967
svm	0.9989	0.9996	0.6156	1.0000	0.7620	0.7615	0.7842	3.6500
lda	0.9989	0.9967	0.6209	1.0000	0.7661	0.7655	0.7875	3.7233

Table 11. *Cont.*

Model	Accuracy	AUC	Recall	Prec.	F1	Kappa	MCC	TT (s)
ridge	0.9988	0.9968	0.5933	1.0000	0.7447	0.7442	0.7698	3.5067
qda	0.9971	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	3.6733
dummy	0.9971	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	3.1667
nb	0.9892	0.0000	0.9755	0.2072	0.3417	0.3386	0.4470	3.5767

6.1. *Arguments for Multinomial Naive Bayes*

The Multinomial Naive Bayes (MNB) classifier demonstrates exceptional performance across various datasets, particularly in terms of accuracy, recall, and training time. Here are the key points supporting the selection of MNB:

- Accuracy and Recall: MNB exhibits high accuracy and recall rates, especially in critical cases where identifying all positive instances is crucial. For instance, in the Android dataset (Table 8), MNB achieved an accuracy of 98.87% and a recall of 56.21%. This indicates that MNB is highly effective in correctly identifying positive instances.
- Training Time: MNB consistently shows the lowest training times across all datasets. This efficiency is crucial for the self-healing system, which requires rapid model updates and retraining. For example, in the Windows dataset (Table 11), MNB's training time was significantly lower than other classifiers, making it an ideal choice for real-time applications.
- Stability and Consistency: MNB maintains stable performance across different datasets, demonstrating its robustness and generalisability. The classifier consistently shows competitive precision and F1-scores, ensuring a balanced trade-off between precision and recall.

We utilised PyCaret [32] for our model comparison analysis and generating the analysis result tables. PyCaret is an open-source, low-code machine-learning library in Python that automates the machine-learning workflow. It allows users to prepare data, compare models, and tune hyperparameters with minimal coding, efficiently handling the entire model comparison process.

6.2. *Best Four Performing Models*

6.2.1. *Android Dataset Results*

As shown in Figure 5, the performance metrics for the best four models on the Android dataset are as follows:

- Naive Bayes: Exhibits perfect recall (100%) but has the lowest precision (9.94%) and accuracy (10.13%), indicating a tendency to over-predict positive instances.
- Quadratic Discriminant Analysis: Shows the highest precision (99.99%) and accuracy (90.08%) but has no recall (0.00%), failing to identify positive instances.
- Logistic Regression: Achieves a balanced performance with an accuracy of 96.03% and a recall of 60.07%, complemented by a high precision of 99.87%.
- SVM—Linear Kernel: Delivers a strong AUC of 99.03%, with good overall accuracy (96.04%) and balanced recall (60.07%) and precision (100%).

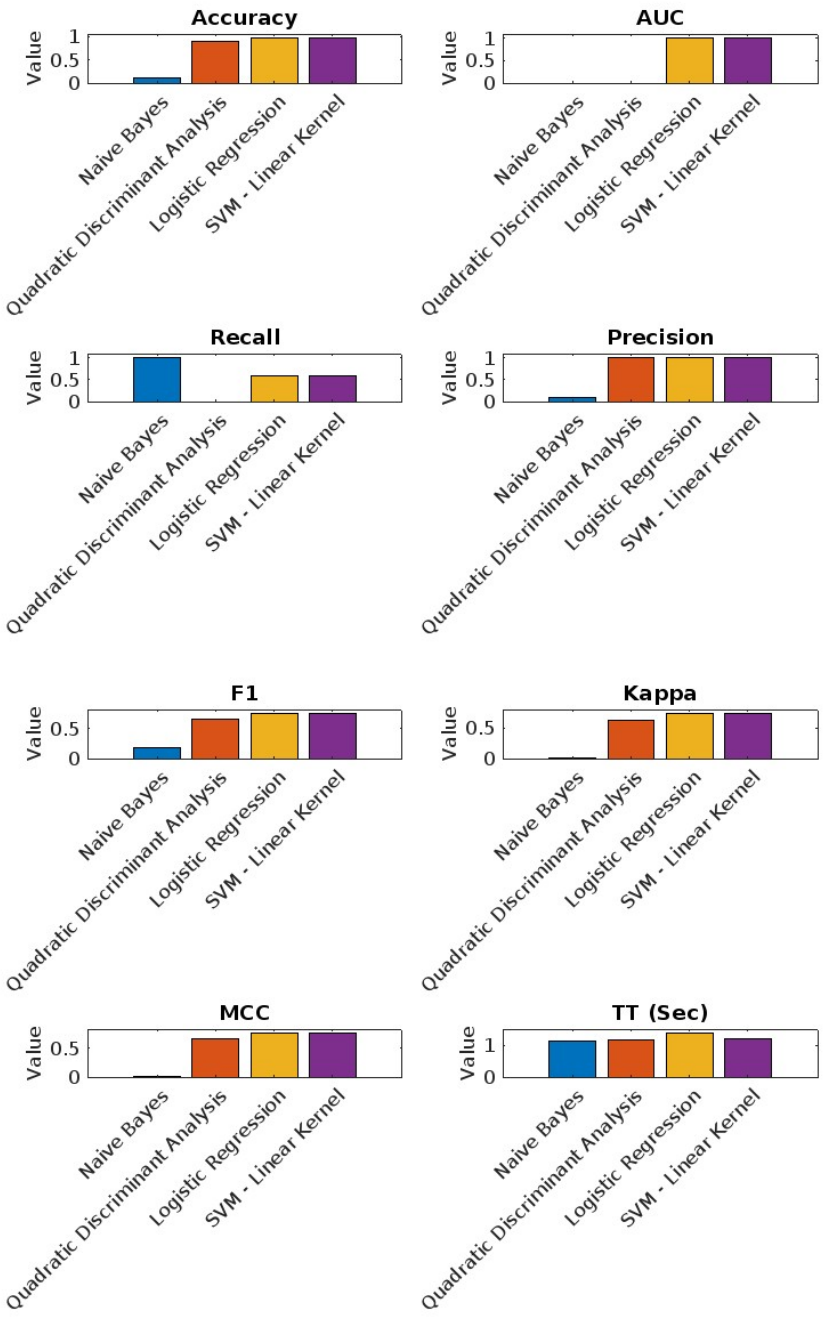

Figure 5. Performance metrics for the best four models in the Android dataset.

6.2.2. *Linux Dataset Results*

As shown in Figure 6, the performance metrics for the best four models on the Linux dataset are as follows:

- Naive Bayes: Highest accuracy (98.87%) and a recall of 56.21%, showing strong performance in identifying positive instances.
- Quadratic Discriminant Analysis: Consistently high metrics with the lowest recall (0.00%).
- Logistic Regression: Good balance between accuracy (98.49%) and precision (90.00%).
- SVM—Linear Kernel: Shows solid performance with high AUC (91.07%).

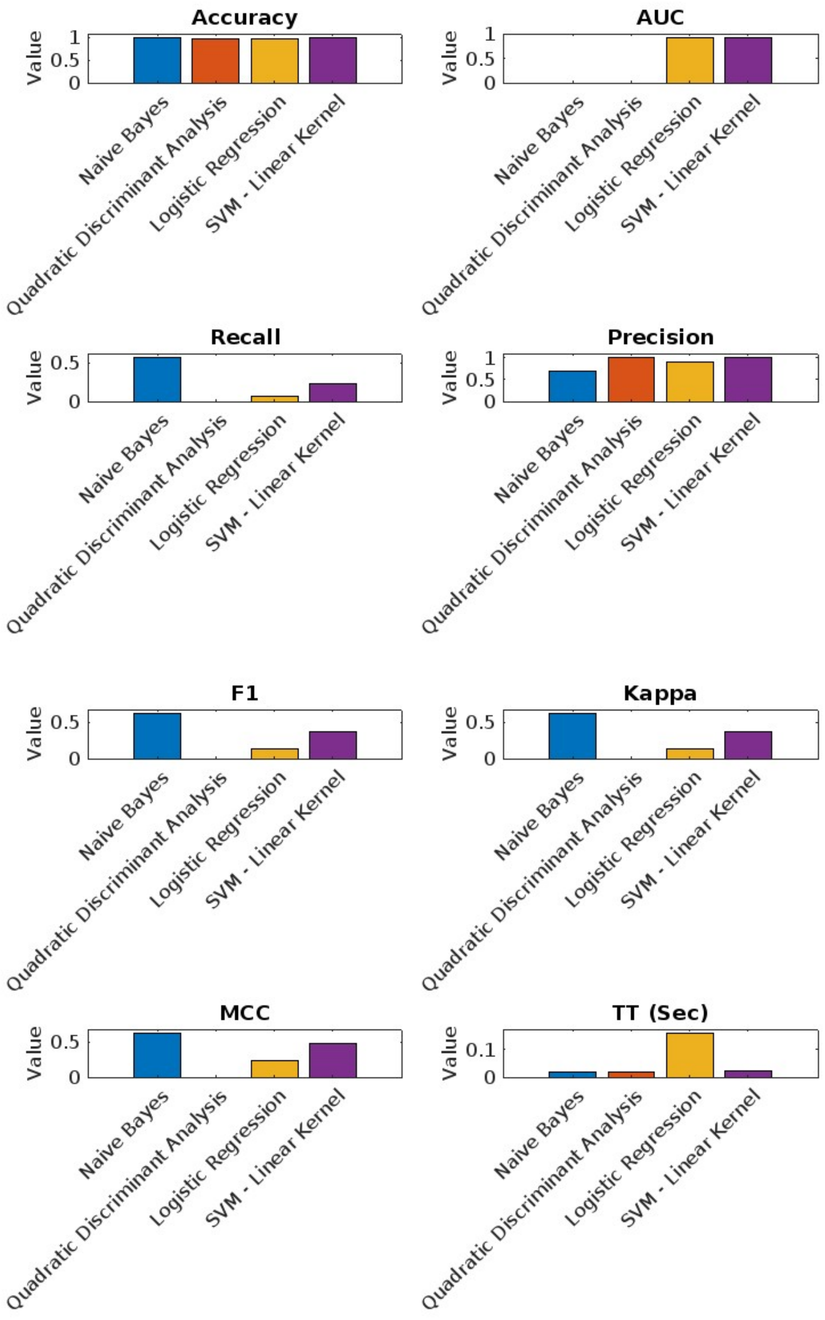

Figure 6. Performance metrics for the best four models in the Linux dataset.

6.2.3. *Mac Dataset Results*

As shown in Figure 7, the performance metrics for the best four models on the Mac dataset are as follows:

- Logistic Regression: Achieved the highest accuracy (94.00%) and AUC (97.14%) with a recall of 61.47%.
- SVM—Linear Kernel: Showed strong performance with high AUC (97.31%) and precision (100%).
- Quadratic Discriminant Analysis: Balanced performance but with lower recall (0.00%).
- Naive Bayes: Perfect recall (100%) but lower accuracy (15.88%).

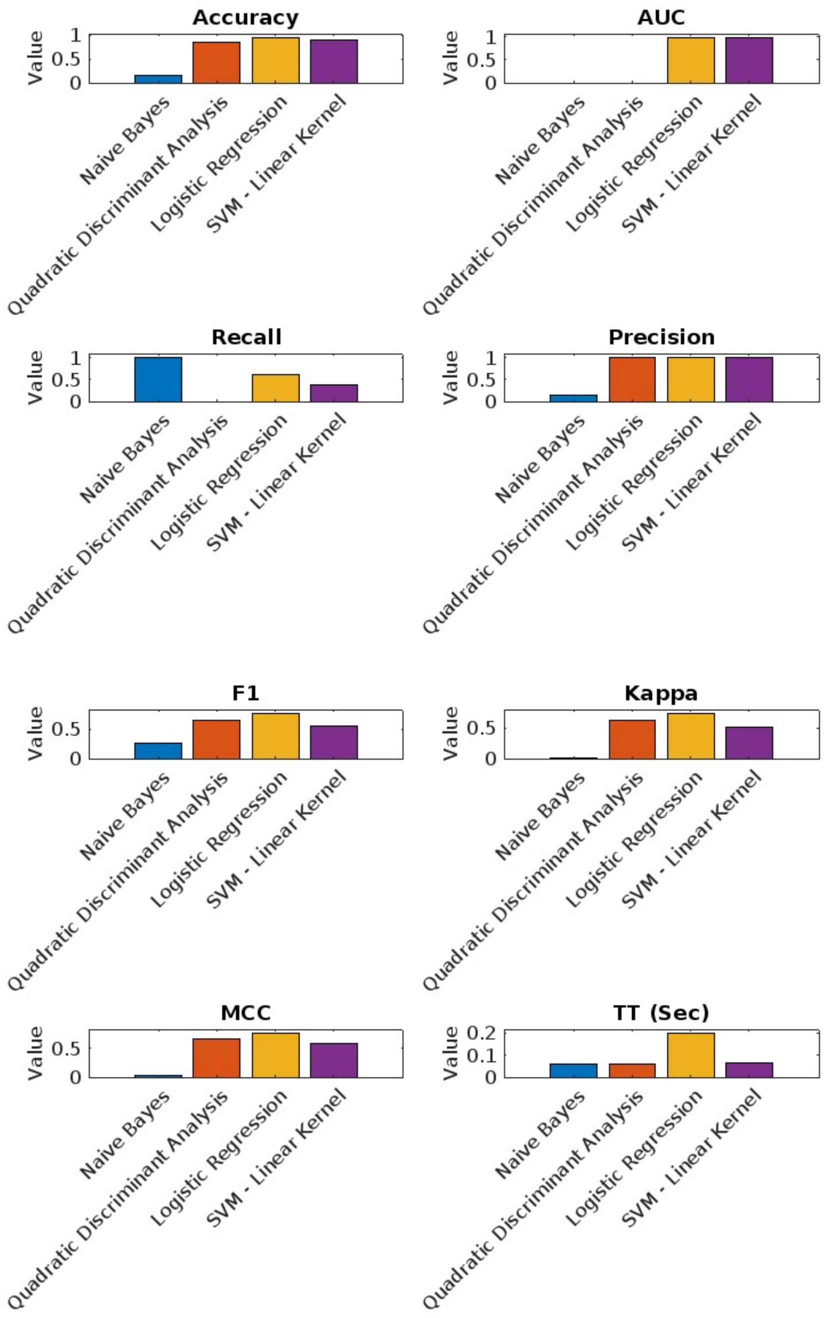

Figure 7. Performance metrics for the best four models in the Mac dataset.

6.2.4. *Windows Dataset Results*

As shown in Figure 8, the performance metrics for the best four models on the Windows dataset are as follows:

- Naive Bayes: Demonstrated high accuracy (97.25%) with a recall of 55.67%, indicating robust performance in identifying positive instances.
- Quadratic Discriminant Analysis: Consistently high performance but with lower recall (0.00%).
- Logistic Regression: Balanced accuracy (96.80%) and precision (89.50%).
- SVM—Linear Kernel: Strong overall performance with high AUC (90.83%).

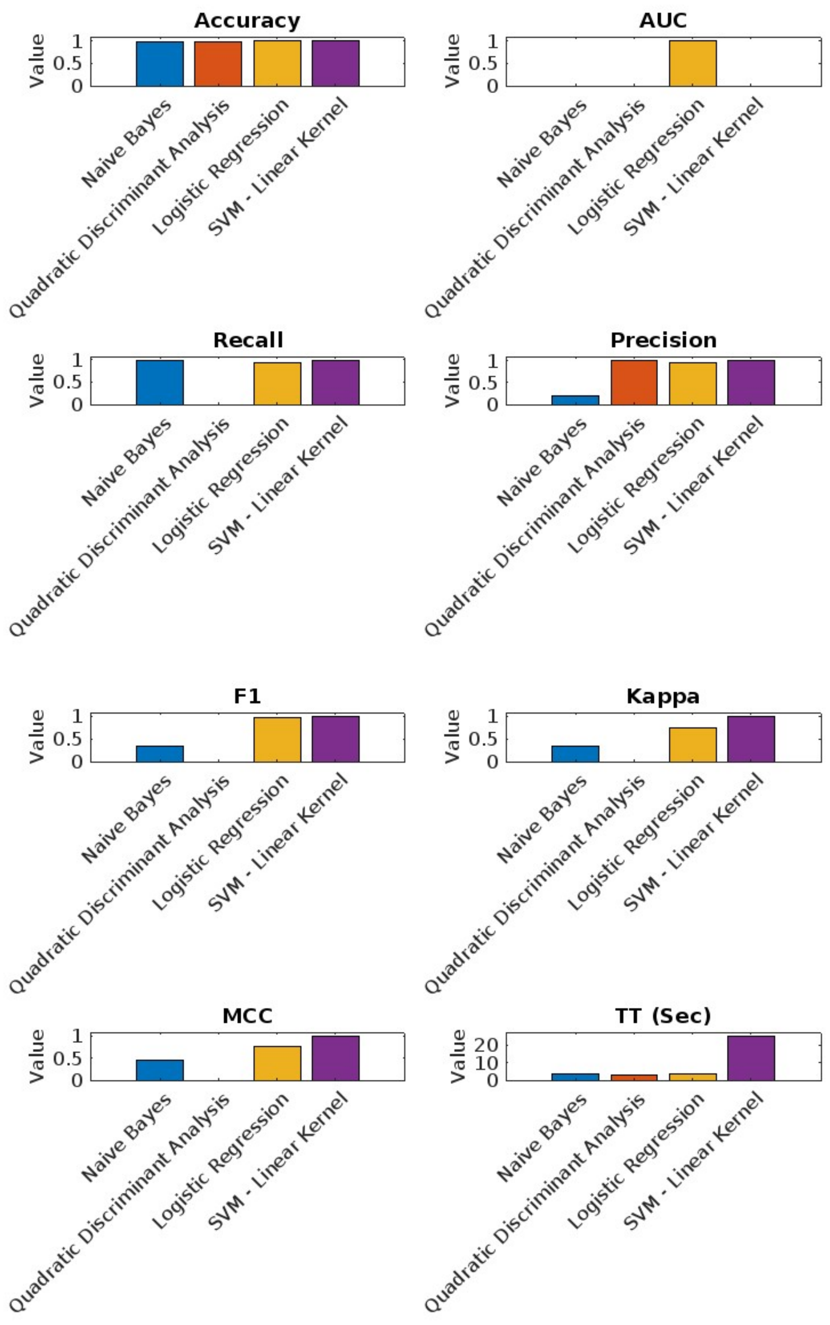

Figure 8. Performance metrics for the best four models in the Windows dataset.

For presenting the best four models' analysis results in a bar chart, we utilised MATLAB [35]. MATLAB is a high-level language and interactive environment used for numerical computation, visualisation, and programming. It allows for sophisticated data visualisation, making it an excellent tool for presenting and interpreting complex data results visually.

7. Comparison with Existing Methods

The comprehensive model evaluation across multiple systems distinguishes this study from existing methods, which often focus on a single model or system. By leveraging PyCaret's capabilities, we were able to conduct a broad comparison of models, revealing insights into their strengths and limitations in different contexts. This approach offers a more nuanced understanding of model performance, suggesting that a combination of models might be necessary to effectively monitor and predict system health across various operating environments. This study's comprehensive comparison highlights the potential of leveraging traditional algorithms such as Multinomial Naive Bayes (MNB) in modern system health analysis tasks. Contrary to the focus on increasingly complex models, this research underscores the value of revisiting classical machine learning approaches. Such comparisons enrich the toolkit available to developers and researchers, offering efficient and scalable solutions to system monitoring challenges. Addressing system log classification within the broader scope of computational self-healing has presented various challenges. These challenges have driven the development of diverse solutions, methodologies, and algorithms aimed at addressing them [34]. As computational self-healing and log classification continue to evolve, a range of traditional and innovative methodologies remain relevant, each offering distinct strengths and challenges. Achieving an effective balance between efficacy, efficiency, and practicality is crucial for practitioners in this field. The multitude of approaches reflects the multifaceted nature of the challenges and the different avenues researchers are exploring to tackle them. We provide a detailed comparison of two pivotal solutions identified in the background work: log-based anomaly detection techniques utilising unsupervised learning and the application of CountVectorizer with Multinomial Naive Bayes for self-healing systems.

7.1. Comparison with ReLoop2

Comparing our Multinomial Naive Bayes (MNB) model for self-healing systems with the ReLoop2 model by [36], we highlight key differences in model performance and objectives. Both models aim to enhance system adaptation, but with different focuses. Our MNB model prioritises error detection and quick retraining, achieving high recall (56.21%) and low training time across multiple datasets, such as Android and Linux. In contrast, ReLoop2 focuses on enhancing recommendation systems by integrating an error compensation loop, which led to an AUC improvement of up to 5.6% on the AmazonElectronics dataset and 3.1% on the MicroVideo dataset. ReLoop2's ability to handle dynamic data distribution shifts without requiring additional model retraining makes it ideal for recommendation tasks, whereas MNB excels in detecting system faults in real-time environments.

Key Comparison Points with ReLoop2

In Table 12, we present a comparison between our MNB model and ReLoop2, with the following key observations:

1. Model Objective:
    - Our MNB model is designed for log-based error detection and self-healing in cyber-physical systems, prioritising high recall and quick training time to identify and correct system faults in real time.
    - ReLoop2 focuses on self-adaptive recommendation systems by integrating error compensation loops to quickly adapt to shifting data distributions in large-scale recommendation datasets like AmazonElectronics and production systems (e.g., Huawei's news feed).
2. Datasets:
    - Our study uses system logs (e.g., Android, Linux, Mac, and Windows logs) for error detection, evaluating performance through metrics like accuracy, recall, and precision.
    - ReLoop2 uses recommendation datasets (AmazonElectronics, MicroVideo, KuaiVideo) and a production dataset, which are much larger and involve user–item interactions, making the task more focused on recommendations.
3. Performance Metrics:
    - Our Model: MNB achieved high recall (56.21%) and balanced accuracy with fast training times, making it suitable for real-time error detection in dynamic environments.

- ReLoop2: Integrating ReLoop2 with baseline models outperformed state-of-the-art methods (e.g., DeepFM, AutoInt+), achieving up to a 5.6% improvement in AUC on the AmazonElectronics dataset. ReLoop2 rapidly compensates for data distribution shifts without requiring additional model training, essential for real-time recommendation systems.

4. Error Handling:
   - MNB: Focuses on identifying system errors through log analysis, excelling in high recall for detecting system warnings and errors.
   - ReLoop2: Incorporates an error memory module that compensates for prediction errors and adapts to changes in data distribution in recommendation tasks, outperforming incremental training techniques in many cases.
5. Training Time:
   - MNB demonstrated significantly lower training times (e.g., 0.019 s for Linux), critical for real-time, self-healing systems.
   - ReLoop2 outperformed incremental training methods without requiring retraining, making it highly efficient for adaptive recommendations in real-time systems.

Table 12. Comparison of MNB and ReLoop2 models.

Model	Dataset	Key Metric	Performance	Key Feature
MNB (Our Work)	Android/Linux	Recall	56.21%	High recall, fast training time (0.019 s).
ReLoop2	AmazonElectronics	AUC improvement	+5.6%	Error memory for adaptive compensation.
	MicroVideo	AUC improvement	+3.1%	Model-agnostic adaptation to changing data.
	Production (Huawei)	AUC	75.0%	Combines error memory and incremental learning.

7.2. Comparison of Machine Learning Models

- Support Vector Machine (SVM): Known for its effectiveness in high-dimensional spaces, SVM can be particularly useful for text classification tasks, such as log analysis.
- Random Forest (RF): This ensemble learning method combines multiple decision trees to improve classification accuracy and robustness, making it suitable for complex datasets like OS logs.
- Gradient Boosting Classifier (GBC): GBC is another ensemble technique that sequentially builds models, optimising for classification tasks by focusing on errors made by previous models.
- AdaBoost (ADA): A boosting algorithm that combines weak learners to form a strong classifier, which can enhance classification accuracy in various scenarios.
- LightGBM: A gradient boosting framework that is optimised for efficiency and speed, particularly when dealing with large datasets.
- Logistic Regression (LR): A baseline model often used for binary and multiclass classification, providing a point of comparison for more complex models.
- Deep Learning Models: While not included in the current comparison due to the scope of our research, we recognise their importance and plan to explore these in future work.

7.3. Experimental Results

Tables 8–11 provide a detailed comparison of the abovementioned models across multiple OS datasets (Android, Linux, Mac Windows). Each model was evaluated based on key performance metrics, including accuracy, AUC, recall, precision, F1-score, kappa, MCC, and training time (TT). Our findings indicate that while models like SVM and Random Forest offer competitive accuracy and AUC, the CountVectorizer coupled with the Multinomial Naive Bayes algorithm demonstrates a unique balance of precision and training efficiency, particularly in scenarios with constrained computational resources.

7.4. Discussion of Model Performance

- Accuracy and Precision: The Naive Bayes model excels in maintaining high accuracy and precision across the datasets, which is crucial for real-time self-healing systems that require immediate and reliable log classification.
- Training Time: The Naive Bayes algorithm also outperforms others in terms of training time, making it a preferred choice for systems where quick model updates are necessary.
- Recall and F1-Score: While some models like SVM and GBC show higher recall, indicating a better ability to detect anomalies, they may require more computational resources and longer training times.

7.5. Performance Across Operating Systems

- Windows Dataset: The Naive Bayes algorithm achieved a recall of 0.9755, which is among the highest across the models tested. This suggests that the Naive Bayes model is highly effective at identifying nearly all instances of errors and warnings in the Windows logs. This high recall is particularly advantageous in a self-healing system where missing even a small percentage of errors could lead to system vulnerabilities or downtime.
- Linux Dataset: On the Linux dataset, the recall drops significantly to 0.5621. This indicates that while the model is still capturing over half of the relevant instances, it may be missing a considerable portion of the anomalies. This decrease in recall could be due to the nature of the Linux logs, which may present a more diverse set of features that are not as well captured by the Naive Bayes model. The complexity and variability in Linux system logs might require additional feature engineering or the integration of more complex models to improve recall.
- Mac Dataset: For the Mac dataset, the Naive Bayes model's recall is 0.6147. While this is better than its performance on the Linux dataset, it still indicates room for improvement. The recall score suggests that the model is moderately successful in identifying relevant log entries but may still miss some critical cases. Given the Mac OS's unique structure and log generation patterns, additional preprocessing or model adjustments could be necessary to enhance recall further.
- Android Dataset: On the Android dataset, the recall achieved by Naive Bayes is 1.0000, which is perfect. This exceptional performance indicates that the Naive Bayes model was able to correctly identify every relevant instance in the Android logs. This level of recall is particularly beneficial in mobile environments where resources are limited, and the cost of errors can be high. A perfect recall means that the model did not miss any errors or warnings, ensuring comprehensive monitoring and quick corrective actions in a self-healing context.

7.6. Interpretation and Implications

- Variability Across Datasets: The Naive Bayes model's recall performance varies significantly across different operating systems. This variability highlights the model's sensitivity to the nature of the data it is applied to. The perfect recall in the Android dataset demonstrates the model's potential when the features are well aligned with the model's assumptions. In contrast, the lower recall on the Linux and Mac datasets suggests that these environments may have more complex log patterns that Naive Bayes alone may not capture fully.

- Advantages in Specific Contexts: The high recall in Windows and perfect recall in Android logs make Naive Bayes a strong candidate for environments where complete anomaly detection is critical, and missing an error could lead to significant consequences. However, in more complex environments like Linux, where the logs may exhibit a wider range of behaviours, relying solely on Naive Bayes may not be sufficient, and combining it with other models or enhancing feature extraction could be necessary.
- Balancing Precision and Recall: It is also important to note the trade-off between precision and recall. While Naive Bayes achieves high recall in certain datasets, precision, particularly in the Windows dataset, is also strong. This balance suggests that Naive Bayes is not only good at catching most of the anomalies but also at ensuring that the identified instances are indeed relevant, reducing the rate of false positives.

7.7. Comparing Different Variants of Naive Bayesian Algorithms

Naive Bayesian algorithms are among the most crucial classifiers used in predictive modelling. These classifiers rely on probability and are based on the general assumption that all features are independent of each other, although this assumption often does not hold in real-world applications. There are three primary variants of Naive Bayes, each with different assumptions that impact their efficiency and accuracy for specific tasks [37]:

1. Multinomial Naive Bayes (MNB): This variant is particularly advantageous for text classification tasks and datasets with discrete features, such as word counts or frequencies. It is highly effective when features represent categorical data.
2. Gaussian Naive Bayes (GNB): GNB is suited for continuous features and assumes that the data follow a normal (Gaussian) distribution, making it ideal for datasets with real-valued features.
3. Bernoulli Naive Bayes (BNB): This variant is best suited for binary/boolean features and is particularly useful when the dataset is composed of binary variables, making it effective in scenarios like text classification where the presence or absence of a feature (e.g., a word) is important.

For our multiple OS log classification and self-healing implementation, we have chosen Multinomial Naive Bayes (MNB). This decision is justified by the nature of system logs, which typically involve categorical data, such as discrete events, counts, and frequencies. MNB effectively classifies and analyses log patterns by leveraging the occurrence of specific events or terms, which is crucial for detecting anomalies and triggering self-healing actions. Gaussian Naive Bayes, in contrast, is more suited for continuous data, which are less common in system logs. This choice aligns with the findings [37] in their work on performance comparison and implementation variants for network intrusion detection, where different Naive Bayesian variants were evaluated for their suitability in various contexts [37].

7.8. Challenges of Manual System Log Analysis

The rich information and pervasiveness of system logs enable a wide range of system management and diagnostics tasks, such as analysing application security, identifying performance anomalies, and diagnosing errors and crashes [38]. When the system in concern has a problem, the corresponding system log entries follow a certain pattern [38], which we are able to exploit to implement anomaly detection using a Multinomial Naive Bayes ML algorithm to detect the anomaly and trigger self-healing remediation. System logs can be used to diagnose the source of the system problems as well as predict potential system issues and problems. The detection of abnormal events in system logs is imperative in real-world cybersecurity operations such as detection, diagnosis, and remediation [38]. Our research specifically addresses the challenge of manually analysing large volumes of system logs, which is a process that is often cumbersome, time-consuming, and prone to errors, particularly in complex environments. Self-healing systems autonomously detect, diagnose, and correct anomalies, with effective self-healing relying heavily on the accurate interpretation of system logs generated by operating systems (OSs). Despite the growing adoption of automated log analysis methods, there are still several contexts in which manual log analysis is prevalent today:

- Legacy Systems: Organisations that continue to rely on older or legacy systems may not have fully integrated automated log analysis tools. In such environments, system administrators and IT support staff often perform manual log analysis to diagnose issues, identify security threats, or troubleshoot system performance.
- Small and Medium-Sized Enterprises (SMEs): Many SMEs may lack the budget to invest in advanced automated log analysis tools. Consequently, IT personnel in these organisations often resort to manual log analysis, particularly during critical incidents or when automated tools fail to provide clear insights.
- Compliance and Auditing: In certain industries, manual log analysis remains a requirement as part of compliance and auditing processes. IT auditors may manually review logs to ensure systems comply with regulatory standards, particularly in sectors such as finance, healthcare, and government.
- Security Operations Centres (SOCs): While SOCs increasingly use sophisticated Security Information and Event Management (SIEM) systems, there are still instances where security analysts perform manual log analysis. This typically occurs when investigating complex or nuanced security incidents that automated tools may not fully capture or interpret.
- Incident Response: During the incident response process, particularly in real-time scenarios, security professionals and incident responders might manually analyse logs to quickly understand and contain an issue before escalating to automated processes.

Our research introduces an intelligent methodology for creating self-healing systems for OSs, focusing on log classification using CountVectorizer and the Multinomial Naive Bayes algorithm. This approach involves preprocessing OS logs to ensure quality, converting them into a numerical format with CountVectorizer, and then classifying them using the Naive Bayes algorithm. The system classifies multiple OS logs into distinct categories, identifying errors and warnings. We tested our model on logs from four major OSs: Mac, Android, Linux, and Windows, sourced from Zenodo to simulate real-world scenarios. The model's accuracy, precision, and reliability were evaluated, demonstrating its potential for deployment in practical self-healing systems. These contexts underscore the ongoing relevance of manual log analysis in specific scenarios, despite the advantages of automation. By automating the log analysis process, as proposed in our research, we aim to reduce the dependency on manual analysis, thereby enhancing efficiency and minimising the likelihood of human error.

7.9. Comparison Between DA-Parser and CountVectorizer

DA-Parser and CountVectorizer are both prominent tools in the field of natural language processing (NLP), yet they serve distinct purposes and are applied in different contexts. Understanding their comparative strengths and limitations is essential for selecting the appropriate tool for specific text analysis tasks. DA-Parser [39] is a specialised tool designed for dialogue act recognition, a task that involves identifying the communicative function of an utterance within a conversation. This parser is particularly adept at parsing dialogue structures, categorising utterances based on their roles in the conversation, such as questions, statements, or requests. The DA-Parser leverages linguistic features that are integral to understanding the context and flow of dialogues, making it a critical tool in conversational analysis, particularly in systems that need to interpret and respond to human communication. In contrast, CountVectorizer is a more general-purpose tool used to convert text documents into a matrix of token counts. This method of feature extraction is foundational in many text classification tasks, where the primary goal is to represent the text in a numerical format that can be fed into machine learning models. Unlike DA-Parser, CountVectorizer does not specialise in any specific type of text or context but is broadly applicable across various text analysis tasks, such as sentiment analysis, document classification, and spam detection.

One of the key differences between these tools lies in their handling of linguistic features. DA-Parser [39] is highly specialised and considers the sequence and role of words within a dialogue, making it more sophisticated in contexts where understanding the function of each utterance is crucial. CountVectorizer, on the other hand, focuses purely on the frequency of tokens, without regard to their sequence or contextual meaning. This simplicity allows CountVectorizer to be widely applicable but also limits its effectiveness in tasks that require a deeper understanding of linguistic context. The output generated by these tools reflects their differing purposes. DA-Parser produces structured outputs that categorise dialogue acts based on their communicative role, which is essential in dialogue systems and conversational agents. In contrast, CountVectorizer generates a sparse matrix of token counts that serves as input for further machine learning processes, such as those demonstrated in sentiment analysis tasks. While DA-Parser is indispensable in tasks involving dialogue act recognition and the parsing of conversational data, CountVectorizer remains a versatile and widely used tool in general text classification. The choice between these tools should be guided by the specific needs of the task at hand, with DA-Parser being more suitable for dialogue-intensive applications, and CountVectorizer being an excellent choice for broader text analysis tasks.

8. Conclusions

In this study, we showed that the Multinomial Naive Bayes (MNB) algorithm, when combined with the CountVectorizer, is a promising machine learning tool for the implementation of self-healing mechanisms in cyber-physical systems. While MNB may not always achieve the highest accuracy compared to more complex classifiers such as logistic regression or k-nearest neighbours, it consistently excels in recall—a crucial metric for the identification of system anomalies, particularly errors and warnings that can threaten system integrity. The high recall rates observed across diverse datasets—Android, Linux, Mac, and Windows—demonstrate the robustness of MNB in detecting potential issues. For instance, MNB achieved a recall rate of 56.21% on the Android dataset and consistently identified over 56% of true positive instances in the Linux dataset. This strong recall is critical for self-healing systems, as it reduces the likelihood that critical issues will go unnoticed, enabling timely interventions such as system reboots, resource reallocation, or other corrective actions that maintain operational stability. Although MNB may exhibit modest accuracy in comparison to other classifiers, its ability to reliably detect anomalies is invaluable in environments where missing faults or warnings could lead to significant consequences. Moreover, MNB's low computational demand and rapid retraining time across all datasets make it especially well suited for real-time applications where models need to adapt quickly to evolving conditions.

The contribution of this research extends beyond the experimental results. Our findings support the deployment of MNB as a practical solution for self-healing in cyber-physical systems, with its strengths lying in recall and computational efficiency. While its precision and accuracy may benefit from future enhancements, the foundational results indicate that MNB plays a key role in ensuring the prompt identification of faults, which is crucial for safeguarding system reliability. Several factors could influence the generalisability of our results. One potential threat is the diversity of datasets; while we tested on multiple datasets (Android, Linux, Mac, and Windows), the results may vary with more complex or domain-specific logs. Another issue is the reliance on tokenisation methods such as CountVectorizer, which may not fully capture the semantic relationships within system logs. Future work should aim to address these limitations by exploring more comprehensive datasets and advanced feature extraction methods.

9. Future Work

While this study lays a solid foundation for the use of MNB in self-healing systems, there are several opportunities for extending this research. Firstly, integrating more sophisticated feature engineering techniques could allow MNB to capture deeper and more complex relationships in system log data, potentially improving its performance in terms of both precision and recall. Expanding the dataset to include more diverse and complex log types from a broader range of systems could help validate the generalisability of our findings, ensuring that the MNB model remains effective in a variety of real-world scenarios. Future work should also explore the potential of combining MNB with other machine learning models in a hybrid, multilayered framework, leveraging the strengths of multiple algorithms. Such an ensemble approach could balance the high recall of MNB with the precision and accuracy of more complex models, creating a more comprehensive self-healing system. Implementing real-time analytics and adaptive learning frameworks could allow for continuous model improvement. This would enable the system to dynamically update in response to evolving threats, ensuring resilience and adaptability in the face of new and emerging cyber-physical challenges. The high recall performance demonstrated by MNB in this study positions it as a critical component of self-healing systems. Its ability to reliably detect system anomalies makes it well suited for environments where operational continuity is vital, such as industrial control systems, healthcare monitoring systems, and smart infrastructure. The next phase of research should focus on enhancing this approach to create more robust, real-time self-healing mechanisms that not only improve system reliability but also reduce the costs associated with downtime and failures in cyber-physical environments.

Author Contributions: Conceptualisation, O.J. and A.S.S.; methodology, O.J.; software, O.J.; validation, O.J., A.S.S. and O.K.; analysis, O.J. and A.S.S.; resources, A.S.S., O.K. and M.A.; data curation, O.J.; writing—original draft preparation, O.J.; writing—review and editing, O.J., A.S.S., O.K. and M.A.; visualisation, O.J.; supervision, A.S.S. and O.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: For the source code and the application, the reader can download the related files via the following link: https://github.com/obinnajohnphill/self-healing-ml-app accessed on 1 October 2024.

Acknowledgments: The authors would like to acknowledge the support given by the Cyber Security Research Group (CSRG) at the Department of Computer Science, Nottingham Trent University.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Adeniyi, O.; Sadiq, A.S.; Pillai, P.; Taheir, M.A.; Kaiwartya, O. Proactive Self-Healing Approaches in Mobile Edge Computing: A Systematic Literature Review. *Computers* **2023**, *12*, 63. [CrossRef]
2. Sarker, I.H. Deep learning: A comprehensive overview on techniques, taxonomy, applications and research directions. *SN Comput. Sci.* **2021**, *2*, 420. [CrossRef] [PubMed]
3. Singh, P.; Saman Azari, M.; Vitale, F.; Flammini, F.; Mazzocca, N.; Caporuscio, M.; Thornadtsson, J. Using log analytics and process mining to enable self-healing in the Internet of Things. *Environ. Syst. Decis.* **2022**, *42*, 234–250. [CrossRef]
4. Shahzad, K.; Iqbal, S.; Fraz, M.M. Automated Solution Development for Smart Grids: Tapping the Power of Large Language Models. In Proceedings of the 2023 17th International Conference on Engineering of Modern Electric Systems (EMES), Oradea, Romania, 9–10 June 2023; pp. 1–4.
5. Hassan, S.U.; Ahamed, J.; Ahmad, K. Analytics of machine learning-based algorithms for text classification. *Sustain. Oper. Comput.* **2022**, *3*, 238–248. . [CrossRef]
6. Gan, S.; Shao, S.; Chen, L.; Yu, L.; Jiang, L. Adapting hidden naive Bayes for text classification. *Mathematics* **2021**, *9*, 2378. [CrossRef]
7. Ahmed, T.; Mukta, S.F.; Al Mahmud, T.; Al Hasan, S.; Hussain, M.G. Bangla Text Emotion Classification using LR, MNB and MLP with TF-IDF & CountVectorizer. In Proceedings of the 2022 26th International Computer Science and Engineering Conference (ICSEC), Sakon Nakhon, Thailand, 21–23 December 2022; pp. 275–280.

8. Siddiqui, T.; Mustaqeem, M. Performance evaluation of software defect prediction with NASA dataset using machine learning techniques. *Int. J. Inf. Technol.* **2023**, *15*, 4131–4139. [CrossRef]
9. Coronado, E.; Behravesh, R.; Subramanya, T.; Fernández-Fernández, A.; Siddiqui, S.; Costa-Pérez, X.; Riggio, R. Zero touch management: A survey of network automation solutions for 5G and 6G networks. *IEEE Commun. Surv. Tutor.* **2022**, *24*, 2535–2578. [CrossRef]
10. Xu, Y.; Liu, X.; Cao, X.; Huang, C.; Liu, E.; Qian, S.; Liu, X.; Wu, Y.; Dong, F.; Qiu, C.-W.; et al. Artificial intelligence: A powerful paradigm for scientific research. *Innovation* **2021**, *2*, 100179. [CrossRef] [PubMed]
11. Ghosh, D.; Sharman, R.; Rao, H.R.; Upadhyaya, S. Self-healing systems—Survey and synthesis. *Decis. Support Syst.* **2007**, *42*, 2164–2185. [CrossRef]
12. Donta, P.K.; Sedlak, B.; Casamayor Pujol, V.; Dustdar, S. Governance and sustainability of distributed continuum systems: A big data approach. *J. Big Data* **2023**, *10*, 1–31. [CrossRef]
13. Bhanage, D.A.; Pawar, A.V.; Kotecha, K. IT infrastructure anomaly detection and failure handling: A systematic literature review focusing on datasets, log preprocessing, machine & deep learning approaches and automated tool. *IEEE Access* **2021**, *9*, 156392–156421.
14. Dash, S.; Shakyawar, S.K.; Sharma, M.; Kaushik, S. Big data in healthcare: Management, analysis and future prospects. *J. Big Data* **2019**, *6*, 1–25. [CrossRef]
15. Odeh, A.H.; Odeh, M.; Odeh, N. Using Multinomial Naive Bayes Machine Learning Method To Classify, Detect, And Recognize Programming Language Source Code. In Proceedings of the 2022 International Arab Conference on Information Technology (ACIT), Abu Dhabi, United Arab Emirates, 22–24 November 2022; pp. 1–5.
16. Alvi, N.; Talukder, K.H. Sentiment analysis of Bengali text using CountVectorizer with logistic regression. In Proceedings of the 2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT), Kharagpur, India, 6–8 July 2021; pp. 1–5.
17. Hafeez, S.; Kathirisetty, N. Effects and comparison of different data pre-processing techniques and ML and deep learning models for sentiment analysis: SVM, KNN, PCA with SVM and CNN. In Proceedings of the 2022 First International Conference on Artificial Intelligence Trends and Pattern Recognition (ICAITPR), Hyderabad, India, 10–12 March 2022; pp. 1–6.
18. Vijay, V.; Verma, P. Variants of Naïve Bayes Algorithm for Hate Speech Detection in Text Documents. In Proceedings of the 2023 International Conference on Artificial Intelligence and Smart Communication (AISC), Greater Noida, India, 27–29 January 2023; pp. 18–21.
19. Patel, A.; Meehan, K. Fake news detection on Reddit utilizing CountVectorizer and term frequency-inverse document frequency with logistic regression, MultinomialNB, and support vector machine. In Proceedings of the 2021 32nd Irish Signals and Systems Conference (ISSC), Athlone, Ireland, 10–11 June 2021; pp. 1–6.
20. Wang, P.; Poovendran, P.; Manokaran, K.B. Fault detection and control in integrated energy system using machine learning. *Sustain. Energy Technol. Assess.* **2021**, *47*, 101366. [CrossRef]
21. Kane, K. Finding The Available Website Name By Using Naive Bayes Classification. In Proceedings of the 2022 International Conference on Decision Aid Sciences and Applications (DASA), Chiangrai, Thailand, 23–25 March 2022; pp. 624–629.
22. Singla, T.; Gaur, V.; Misra, D.K. Comparison between Multinomial Naive Bayes and Multi-Layer Perceptron for Product Review In Real Time. In Proceedings of the 2022 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COM-IT-CON), Online, 26–27 May 2022; Volume 1, pp. 374–379.
23. Singh, G.; Kumar, B.; Gaur, L.; Tyagi, A. Comparison between multinomial and Bernoulli naïve Bayes for text classification. In Proceedings of the 2019 International Conference on Automation, Computational and Technology Management (ICACTM), London, UK, 24–26 April 2019; pp. 593–596.
24. McCallum, A.; Nigam, K. A comparison of event models for naive bayes text classification. In Proceedings of the AAAI-98 Workshop on Learning for Text Categorization, Madison, WI, USA, 26–27 July 1998; Volume 752, pp. 41–48.
25. Joachims, T. A probabilistic analysis of the Rocchio algorithm with TFIDF for text categorization. In Proceedings of the Fourteenth International Conference on Machine Learning, Nashville, TN, USA, 8–12 July 1997; Volume 97, pp. 143–151.
26. Manning, C.D.; Raghavan, P.; Schütze, H. Xml retrieval. In *Introduction to Information Retrieval*; Cambridge University Press: Cambridge, UK, 2008.
27. Jurafsky, D.; Martin, J.H. Spelling Correction and the Noisy Channel. In *Speech and Language Processing*, 2nd ed.; Prentice-Hall: Upper Saddle River, NJ, USA, 2009.
28. Panda. Available online: https://pandas.pydata.org/ (accessed on 28 September 2023).
29. NumPy. Available online: https://numpy.org/ (accessed on 28 September 2023).
30. Seaborn. Available online: https://seaborn.pydata.org/ (accessed on 28 September 2023).
31. Scikit-learn. Available online: https://scikit-learn.org/stable/ (accessed on 25 September 2023).
32. PyCaret. Available online: https://pycaret.org/ (accessed on 29 September 2023).
33. Tabulate. Available online: https://pypi.org/project/tabulate/ (accessed on 27 September 2023).
34. Soldani, J.; Brogi, A. Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey. *ACM Comput. Surv. (CSUR)* **2022**, *55*, 1–39. [CrossRef]
35. MathWorks. MATLAB—MathWorks. Available online: https://matlab.mathworks.com/ (accessed on 28 May 2024).

36. Jieming, Z.; Guohao, C.; Junjie, H.; Zhenhua, D.; Ruiming, T.; Weinan, Z. ReLoop2: Building Self-Adaptive Recommendation Models via Responsive Error Compensation Loop. In Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Long Beach, CA, USA, 6–10 August 2023; pp. 5728–5738.
37. Ige, T.; Kiekintveld, C. Performance Comparison and Implementation of Bayesian Variants for Network Intrusion Detection. In Proceedings of the IEEE International Conference on Artificial Intelligence, Blockchain, and Internet of Things (AIDThings), Mount Pleasant, MI, USA, 16–17 September 2023.
38. Zhou, J.; Qian, Y.; Zou, Q.; Liu, P.; Xiang, J. Deepsyslog: Deep Anomaly Detection on Syslog Using Sentence Embedding and Metadata. *IEEE Trans. Inf. Forensics Secur.* **2022**, *17*, 3051–3061. [CrossRef]
39. Liu, Y.; Tao, S.; Meng, W.; Wang, J.; Hao, Y.; Jiang, Y. Multi-Source Log Parsing with Pre-Trained Domain Classifier. *IEEE Trans. Netw. Serv. Manag.* **2024**, *21*, 2651–2663. [CrossRef]