Logminder

Jatin Subhashchandra Jangir, 241110031, jatinj24@iitk.ac.in
Milan Roy, 241110042, milanr24@iitk.ac.in
Ananya Chaturvedi, 241110601, ananyac24@iitk.ac.in
Ritam Acharya, 210859, ritam21@iitk.ac.in

CS685: Data Mining

Abstract

Logminder is an automated system built to identify and detect anomalies within system logs, targeting unusual patterns that may indicate potential issues. Complex systems generate vast amounts of log data containing detailed information on events, errors, and other critical activities that reflect the system's health and behavior. Due to the sheer volume and complexity of these logs, manual analysis is both impractical and inefficient. Logminder addresses this challenge by implementing log parsing and semantic embedding to prepare and process log data for automated analysis. By detecting anomalies promptly, Logminder enables proactive issue resolution, improving overall system reliability and performance. This approach not only enhances operational efficiency but also ensures that potential problems are addressed before they can escalate into significant failures, providing a robust, scalable solution for managing large-scale log data in complex systems.

Soucre Code: https://github.com/jatin-jangir/logminder

1 Motivation of the Problem

In modern industries like finance, healthcare, and e-commerce, complex systems form the backbone of essential operations, impacting millions of users daily. As these systems become complex, they generate an increasing volume of log data, recording every event, error, and operational detail. This log data is invaluable, offering insights into system performance, behavior, and potential issues. However, the sheer volume and intricacy of logs create significant challenges for manual analysis. Analyzing logs to detect patterns or anomalies manually is time-consuming and inefficient, risking delays that can lead to operational disruptions, user dissatisfaction, and financial losses. Minor issues in system logs, if left unaddressed, can escalate into major problems, including service outages, data integrity breaches, and security vulnerabilities, directly impacting user experience, customer trust, and overall financial outcomes. Given these challenges, an automated log-based anomaly detection system like Logminder becomes essential for maintaining system stability. Logminder's capability to identify unusual or abnormal patterns in real time provides timely alerts, allowing for swift intervention and minimizing potential disruptions. Automation in log analysis not only enhances the speed and accuracy of detecting issues but also frees up system administrators to focus on higher-value tasks beyond routine troubleshooting. Through the integration of semantic embedding and machine learning-driven anomaly detection, Logminder can process logs efficiently and recognize patterns that may go unnoticed in manual analysis. This is particularly useful in environments where the high volume of data necessitates a scalable and efficient solution. The development of Logminder is driven by the need to improve system reliability and performance, reduce the risk of unexpected failures, and enhance overall operational efficiency. By continuously learning from live logs and adapting to emerging patterns, Logminder supports agile problem-solving and long-term system improvement, making it an essential tool for managing the integrity and resilience of complex systems.

2 Data Used

We have used the **HDFS (Hadoop Distributed File System)** dataset for this project as it is widely used for studying log anomaly detection and because the dataset is fully labeled. It consists of system logs generated by a Hadoop cluster, often used to benchmark models that detect anomalies in large-scale distributed systems. The dataset is composed of log files from HDFS, capturing system events like file creations, replications, deletions, and error events within a Hadoop cluster.

2.1 Key Characteristics of the HDFS Dataset

- **Log Events/Lines**: Each line in the dataset represents a single log event or action within the HDFS system. These lines contain various attributes, including timestamps, log levels (`INFO`, `WARN`, `ERROR`), source hosts, block IDs, and messages describing the event.
- **Block IDs**: Each log line associated with a data block is assigned a block ID, which identifies a unique data block within the system. This ID is crucial for grouping events related to the same data block, enabling the identification of patterns or unusual behaviors.
- **Labels**: The dataset is fully labeled, indicating whether each block is ”normal” or ”anomalous”.
- **Templates**: The dataset is also accompanied by a list of templates used for generating the log lines.

2.2 Statistics of the HDFS Dataset

- **Number of Log Events**: 11175629
- **Number of Unique Block IDs**: 575061
- **Number of Normal Blocks**: 558223
- **Number of Anomalous Blocks**: 16838

2.3 Test Set Statistics

- **Total Number of Blocks**: 115,013
- **Number of Normal Blocks**: 111,645
- **Number of Anomalous Blocks**: 3,368
- **Ratio of Normal Blocks**:

$$\frac{111,645}{115,013} = 0.9707163538034831$$

- **Ratio of Anomalous Blocks**:

$$\frac{3,368}{115,013} = 0.029283646196516914$$

3 Initial Approach

This project idea is primarily inspired by the paper "Semi-supervised Log-based Anomaly Detection via Probabilistic Label Estimation."[3] The paper addresses a key challenge in training log anomaly detection models—the scarcity of labeled data. Typically, logs are labeled manually by domain experts, a process that is both labor-intensive and time-consuming. To overcome this challenge, the paper proposes a Probabilistic Label Estimation technique. This method leverages clustering to estimate the labels for unlabeled data when a limited amount of labeled data is available.

For log parsing, the authors utilized the Drain parser[4] to process log messages. System-generated logs often consist of complex, unstructured data with recurring patterns, making direct analysis challenging. The Drain parser simplifies this by converting raw log messages into structured templates. It achieves this by grouping similar log lines and extracting patterns to generate log templates.

3.1 Problem

While attempting to implement the aforementioned parsing technique for our project, we faced a challenge. Drain is built to extract and generate templates from unstructured log data, rather than working with predefined templates. This posed an issue for us, as our training and testing environments are kept separate. To simulate a real-world log generation environment, the program must rely solely on the trained model during testing, with the raw log messages as input—rather than the preprocessed data. This requires the testing program to preprocess the log messages in the same way as during training. Although we could still use DRain to parse the logs and generate templates, there would be no assurance that the templates generated during testing would match those produced during training. Moreover, since DRain uses clustering to create templates, it becomes time-consuming and impractical for use in a real life setting.

3.2 Soltution

To address this issue, we implemented a solution combining the predefined templates provided with the dataset and regular expressions. Each predefined template was assigned a unique template ID and converted into a corresponding Regex pattern. During log parsing, each log message was matched against these template patterns, and the matching template's ID was assigned to the log message. When constructing vector representations for each block, the template embeddings corresponding to all the templates in the block were summed to create the final representation.

4 Methodology

4.1 Log File Preprocessing

The log events from the log file needed to be preprocessed before they could be used to train or test the model.

4.1.1 Creating the Test Set

Due to the complexity of generating a test set of log files from scratch with labeled data, we opted to split the original log file into training and test sets. To more accurately simulate a real-world log generation environment, we chose to split the raw log file prior to preprocessing, rather than after, which is the more conventional approach.

To prevent data leakage, we ensured that all log events associated with the same block were exclusively assigned to either the train or test set. We structured the split so that older logs were included in the train set, while newer logs populated the test set. This was achieved by initially grouping log events by block IDs.

To retain the original balance of normal and anomalous labels in the train and test sets, we further divided the grouped log events by label after block-based grouping. Because logs often have temporal dependencies—where earlier entries provide context for later anomalies—we aimed to preserve the sequence of log events. Thus, the label-based grouping of blocks was ordered according to the line number of the first event within each block. Blocks were then copied into the train and test sets according to the predefined split ratio for both normal and anomalous labels.

Finally, to replicate real-world conditions where log events from a single block are not printed sequentially, we sorted the log events within the train and test sets by their line numbers in the original raw log file. The CSV file containing the labels for each block is similarly divided to correspond with the train and test splits, ensuring that each set has its associated label information aligned with the log data. This approach preserves label integrity for each block in both the training and testing phases.

4.1.2 Preprocessing the Training Data

The training set log file consists of log events as individual log lines, with multiple log events mapping to a single block. Importantly, log events belonging to the same block may not appear consecutively. These log lines cannot be directly used by the ML model for training and require preprocessing. The purpose of the preprocessing step is to obtain a vector representation for each input, where each block serves as an individual input, so a unique vector representation will be created for each block.

In the HDFS dataset, each log event is derived from a fixed set of log templates. For generating vector representations, we use these log templates rather than the actual log events. Regular expressions are employed to map each log event to a specific block ID and its corresponding template, creating a sequence of templates for each block. The vector representation for each block is created by summing the vector representations of the individual log templates that constitute the block.

To create the vector representations of log templates, **TF-IDF** vectorization was used to generate a TF-IDF matrix. This matrix helps identify the importance of each word within a specific log template relative to all templates in the dataset, ensuring that distinctive and relevant terms are weighted higher. To further capture the semantic meaning and context of the words in the templates, **GloVe** embeddings were incorporated, which provide a dense vector representation for each word. This approach ensures that the model remains robust even when log templates are slightly updated but retain their semantic meaning.

Finally, the vector representation for each log template is generated by multiplying the weight of each word from the TF-IDF matrix with its corresponding word embedding from GloVe and then summing the results across all words in the template. This combined representation incorporates both term importance and semantic context, improving the model's ability to handle variations in log templates while maintaining their underlying meaning.

Due to the tendency of programmers to use *snake_case* or *camelCase* for combining multiple words, these combined words are split after the TF-IDF vectorization process. To address this, the value used to multiply the weight of each word from the TF-IDF matrix is the sum of the GloVe embeddings for each of the individual words into which the original word was split. This approach ensures that the semantic meaning of the combined word, reflected by its components, is properly captured, and the vector representation remains consistent with the intended meaning, even after the word is split.

4.2 Model

We have implemented three different Machine Learning models, namely, **Logistic Regression**, **Decision Trees**, and **Feed Forward Neural Network (FFNN)**.

4.2.1 Logistic Regression

Logistic Regression is a simple yet powerful algorithm, particularly suitable for binary classification tasks, and serves as a baseline model for performance comparison. The model is highly interpretable, as it provides insights into how features contribute to predictions through the learned weights. Logistic Regression is computationally efficient and works well on smaller datasets or datasets with linear relationships.

4.2.2 Decision Trees

Unlike Logistic Regression, Decision Trees can handle non-linear relationships between features and the target variable. Decision Trees naturally rank features based on their importance, offering additional insights into the dataset. They are versatile and can model complex patterns in the data without requiring heavy preprocessing.

4.2.3 Feed Forward Neural Network (FFNN)

Feed Forward Neural Networks (FFNNs) are capable of learning complex, non-linear patterns in the data, making them suitable for datasets where simpler models might struggle. Neural networks can automatically learn interactions between features, which is particularly useful in high-dimensional datasets. With appropriate tuning, FFNNs often achieve better generalization on unseen data compared to simpler models.

4.2.4 Model Comparison Approach

By combining these models, we aimed to explore a range of approaches from simple, interpretable methods to more complex, powerful ones. This allowed us to evaluate their strengths and weaknesses for the specific characteristics of our dataset.

4.2.5 Test Set Analysis

Our initial test set contains approximately 3% anomalous data. To analyze the model's performance on test sets with varying anomaly percentages, we generated test sets with anomalous data ranging from 10% to 90%. The comparison results are illustrated in the Results section.

4.3 Implementation

4.3.1 What is Docker?

Docker is an open-source platform that provides a simple way to package, share, and run applications in isolated environments called containers. A container is a lightweight, standalone unit that includes everything needed to run the application, including the code, runtime, system libraries, and dependencies. In this project, Docker is used to containerize the application, which ensures that it runs consistently across different environments—whether on a developer's local machine or in production. Docker containers are designed to be highly portable and require minimal overhead, allowing for rapid testing and iteration during development [1]. By isolating dependencies, Docker makes the application more resilient to changes in its environment, which is crucial for complex systems. The use of Docker ensures that the project's components are reliable, reproducible, and easy to distribute, which simplifies collaboration between development and operations teams.

4.3.2 What is Kubernetes?

Kubernetes is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. It is responsible for managing clusters of Docker containers and ensuring they run reliably and efficiently. In this project, Kubernetes serves as the orchestration layer, managing how and where containers run, and helping to automate tasks like load balancing, scaling, and failover [2]. With Kubernetes, application components are deployed as a set of pods, which are the smallest deployable units in Kubernetes and consist of one or more containers. Kubernetes automatically manages scaling based on the load—adding or removing instances as necessary to maintain performance. Additionally, Kubernetes supports self-healing mechanisms: if a container crashes, Kubernetes will automatically restart or replace it to ensure minimal disruption. The use of Kubernetes in this project also simplifies deployment to cloud environments, providing the flexibility to utilize cloud resources efficiently. Kubernetes handles networking, storage orchestration, and deployment rollouts, ensuring the application is always available and reliable for end users.

4.3.3 Kubernetes Integration

Setup

Kubernetes API Utilization: In this part of the project, the Kubernetes API is used to directly interact with the cluster to gather real-time log data from running pods. The Kubernetes API client (kubernetes Python client) serves as the bridge that allows programmatic access to different Kubernetes resources.

Access Configuration: The script first tries to load an in-cluster configuration using `config.load_incluster_config()`. This is used when the script is running inside a Kubernetes cluster (e.g., as a Pod). It allows seamless interaction with other services in the same cluster. If the script is not running within the cluster, it falls back to using a local kubeconfig (`config.load_kube_config()`). This enables local testing and development, allowing the script to access the Kubernetes cluster remotely.

Log Retrieval: The Kubernetes API client (`v1 = client.CoreV1Api()`) is used to list pods in a specific namespace by calling `v1.list_namespaced_pod(namespace)`. To fetch logs from a specific container inside a pod, the script uses the `v1.read_namespaced_pod_log()` method, which allows specifying details like pod name, container name, and time range (e.g., since_seconds). This way, the logs can be retrieved efficiently without having to collect redundant data.

Namespace Monitoring

Namespace Selection: The project focuses on monitoring specific Kubernetes namespaces. A namespace is a virtual cluster within the Kubernetes environment, and each namespace is used to isolate workloads. For this project, the namespaces are specified via the environment variable `NAMESPACE`, which may include one or multiple namespaces separated by commas (e.g., default, production, development). The user can choose which namespaces to monitor by setting this environment variable, which helps limit monitoring to relevant sections of the cluster. This is particularly useful for controlling the scope and avoiding unnecessary monitoring of non-critical pods.

Pod and Container Monitoring: The script iterates over each pod in the selected namespaces (`v1.list_namespaced_pod(namespace)`), and for each pod, it iterates over the containers (for `container in pod.spec.containers`). Kubernetes pods can contain multiple containers, and each container may generate distinct logs. This ensures that each container within a pod is monitored for log entries. By doing this, the script can capture specific metrics related to each container's activity, which is useful for more granular analysis (e.g., distinguishing between application and infrastructure-level logs within the same pod).

Dynamic Monitoring: The real-time monitoring of the log entries is achieved by continuously fetching logs at set intervals (e.g., every 60 seconds). This dynamic interaction ensures that the system can detect issues as they happen, instead of relying on static log dumps. Since Kubernetes environments are highly dynamic, with pods starting, stopping, or restarting frequently, the namespace and pod discovery process repeats each time the monitoring cycle begins. This helps keep track of the current cluster state without manual intervention.

4.3.4 What can we do from logs ?

Logs of containers are crucial for monitoring, debugging, and maintaining the health of the application. In a containerized environment, logs provide detailed insights into the behavior of the application, capturing events, errors, and other operational data that occur during runtime. Container logs are essential for diagnosing issues, identifying performance bottlenecks, and ensuring that services are running as expected.

In this project, logging is used to track the activity of each container, enabling the team to quickly detect and resolve issues. By aggregating logs from multiple containers, it is possible to get a holistic view of the entire system's performance and identify patterns that may indicate potential problems. Kubernetes further aids in managing logs by allowing centralized logging solutions, making it easier to collect, store, and analyze logs from all containers in the cluster. This approach ensures that any anomalies or failures are detected early, helping maintain the reliability and stability of the application.

Real-Time Log Monitoring and Analysis Kubernetes API for Log Retrieval: The process of retrieving logs begins with interacting with the Kubernetes cluster using the Kubernetes API client (CoreV1Api). Here's how it works: Pod Discovery: The script starts by listing all the pods within the specified namespace using `(v1.list_namespaced_pod(namespace)`. This method returns a list of pods, allowing the script to identify the pods that are currently running. Container Log Retrieval: For each pod discovered, the script further identifies each container using pod.spec.containers. Each container may produce logs independently, so it's essential to retrieve logs for every container individually. Fetching Logs: To actually retrieve the logs, the script uses the (`v1.read_namespaced_pod_log()`) function, providing parameters such as: Pod Name and Namespace to locate the pod. Container Name to specify which container's logs are to be fetched. Since Time (since_seconds parameter) to fetch logs generated since a specific timestamp. This allows incremental log collection rather than fetching all historical logs repeatedly, which optimizes performance and reduces unnecessary data retrieval. The logs are fetched as plain text and split into individual log lines (logs.splitlines()). Each line represents an event or a log message generated by the container.

Real-Time Nature: The script is set to continuously monitor the cluster, meaning it runs indefinitely in a loop. Every 60 seconds, it repeats the process of retrieving new logs. This ensures that logs are being collected in real-time, providing an up-to-date view of the system's state and enabling prompt anomaly detection.

4.3.5 Pipeline for Analysis

Initial Log Processing: Once the logs are retrieved, they need to be processed to make them suitable for analysis: The raw log data is split into individual log lines. Each line is parsed to check for important keywords like "warn", "error", or "fatal". This helps identify which log entries may indicate issues that require attention. *Feature Extraction*: The script classifies log lines into different categories — specifically counting lines that indicate errors and those that do not. Features like the number of errors vs. non-errors, frequency of certain log levels, or specific error codes are extracted. These features are crucial for the ML model to understand the behavior of the system and detect potential anomalies.

Data Preparation for ML Analysis: After processing and classifying the log lines, the extracted information needs to be formatted in a structure that is suitable for the ML model: The extracted metrics (e.g., error count, non-error count) are formatted as inputs for the anomaly detection model. Depending on the model, the input may include additional contextual features (like time of day or pod characteristics).

Sending Logs to the ML Model: The pre-processed log data is then fed to the ML model, which may be deployed in the same environment (e.g., as a service or as part of a running application). The ML model analyzes the current state of log data to determine if the logs represent an anomaly. For instance, if the number of "error" entries suddenly spikes, the model may flag it as an anomaly based on its training.

Real-Time Analysis: Since the log collection process is continuous, the ML model also continuously receives data in real-time. This real-time processing ensures that anomalies are detected as soon as possible, enabling quick responses to potential issues.

Feedback Loop: Once the ML model analyzes the log data, the results are either used to trigger alerts or stored for further analysis. These results are also stored in PostgreSQL for further analysis, which helps in evaluating trends over time or generating periodic reports. If an anomaly is detected, this may also initiate an automated action, such as alerting the engineering team or scaling up/down certain services.

4.3.6 Storage in PostgreSQL

Definition of Time-Series DataTime-Series Data refers to a series of data points indexed in time order. In the context of system monitoring or anomaly detection, each data point typically consists of measurements taken at successive points in time. This type of data is highly valuable for understanding trends, identifying anomalies, and making forecasts over time. Examples include stock prices recorded every day, temperature

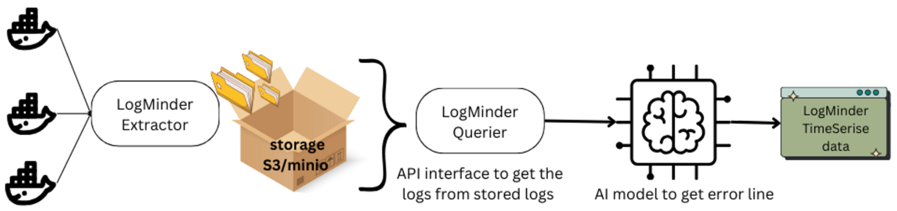
LogMinder Extractor
storage S3/minio
LogMinder Querier
API interface to get the logs from stored logs
AI model to get error line
LogMinder TimeSerise data

Figure 1: Archirecture of Logminder

changes throughout the day, or, in your case, metrics from log files recorded continuously as logs are generated by Kubernetes containers. Storing Results in PostgreSQL The project utilizes a PostgreSQL database to store log metrics that are extracted and analyzed from Kubernetes logs. Here's how the results are structured and stored as time-series data:

1. *Table Structure:*
   The PostgreSQL table used for storing log metrics, named `log_metrics`, is structured to record relevant information from Kubernetes pods in a time-series format. Each row in the table represents a snapshot of metrics for a particular container at a specific point in time. The structure of the table includes the following fields:

   - *timestamp:* A column of type `TIMESTAMP WITH TIME ZONE` that records the exact time at which the metrics were collected. This timestamp is essential for time-series data, as it allows the metrics to be indexed and queried chronologically.
   - *namespace:* The namespace from which the log metrics were collected, allowing metrics to be grouped or filtered by specific sections of the Kubernetes environment.
   - *pod:* The name of the pod that produced the logs.
   - *container:* The name of the container within the pod, which helps identify the specific application or service the metrics are related to.
   - *error_count:* The number of log lines flagged as errors during the given time interval.
   - *non_error_count:* The number of log lines that did not indicate errors.

```sql
CREATE TABLE log_metrics (
    timestamp TIMESTAMP WITH TIME ZONE,
    namespace VARCHAR(255),
    pod VARCHAR(255),
    container VARCHAR(255),
    error_count INTEGER,
    non_error_count INTEGER
);
```

2. *Data Insertion as Time-Series*
   Data Insertion: The script inserts the metrics into the table after each monitoring cycle. During each cycle, a new row is inserted into the `log_metrics` table using the `INSERT` SQL command. This row contains the timestamp at which the logs were analyzed, along with the log metrics

(`error_count` and `non_error_count`) for that period.
Current Time as Reference: When log data is collected and processed, the current timestamp (in UTC format) is used to mark the time at which the metrics are stored, accurately reflecting the data collection time.

In this example:

- The `timestamp` value (`2024-11-13T10:45:00Z`) represents when the logs were analyzed.
- `namespace`, `pod`, and `container` columns provide context on where the log data originated.
- `error_count` and `non_error_count` store the respective metrics collected during this period.

3. *Time-Series Benefits for Analysis and Visualization:*
By storing each log metric with its respective timestamp, the data forms a time-series, which can be used to analyze trends over time. This approach is particularly useful for identifying patterns, such as increasing error rates or recurring warnings during specific periods.

4. *Aggregation and Queries:*
Time-series data allows for efficient aggregation, such as calculating daily error trends, average logs per hour, or comparing error counts across different namespaces over time. PostgreSQL supports various functions that simplify running these types of temporal queries.

5. *Visualization:*
Time-series data is also highly suitable for visualization tools like Grafana or Tableau. The metrics stored in PostgreSQL can be visualized to create dashboards that display changes in error rates over time, identify spikes, and assist in diagnosing potential issues within the Kubernetes environment.

4.3.7 Frontend Part: Log Metrics Dashboard

The frontend of the Logminder project is built using Flask and serves as a web-based dashboard for visualizing log metrics in real-time. This part focuses on providing an interactive interface for users to view aggregated log metrics, making the analysis of Kubernetes container logs accessible and easy to understand. Below are the different components of the frontend methodology:

1. *User Interface Overview:* The web dashboard is designed to provide users with a straightforward view of their Kubernetes log metrics, allowing for easy interaction and visualization of the collected data. The web interface consists of:

2. *Namespace Selection:* A dropdown menu for selecting the namespace from which the user wants to see log metrics.

3. *Container Selection:* A dropdown menu that dynamically updates based on the selected namespace, allowing users to choose the container to visualize.

4. *Time Range Selection:* Another dropdown menu that allows users to select the time range for log analysis (e.g., 10 minutes, 30 minutes, 1 hour, etc.).

5. *Graph Area:* A section of the page dedicated to displaying log metrics in the form of a bar chart using Plotly.

6. Flask Backend for Frontend Communication: Flask Backend for Frontend Communication The Flask application acts as a backend to serve the HTML page and respond to API requests from the frontend.

7. *API Endpoints:*

   (a) `/get_namespaces`: This endpoint is used to fetch the distinct namespaces available in the `log_metrics` table. The namespaces are displayed as options in a dropdown menu, which allows users to select the namespace they are interested in monitoring.

   (b) `/get_containers`: After selecting a namespace, this endpoint fetches the list of containers within the selected namespace. This information is also used to populate a dropdown in the dashboard.

   (c) `/get_metrics`: Once a namespace, container, and time range are selected, this endpoint queries the `log_metrics` table to fetch the log metrics (i.e., `error_count` and `non_error_count`) based on the selected parameters. The query filters data based on:

   - The selected namespace.
   - The selected container.
   - A time limit derived from the current UTC time and the selected time range (e.g., 10 minutes ago, 30 minutes ago).

5 Results

We can observe from confusion matrix of Decision tree 3, FFNN 4 and Logistic regression 4 that the decision trees had the highest TRUE POSITIVE for Anomalies with FFNN following closely behind. Variation of Anomaly to Normal ratio did not affect the any models ability to classify anomalies.

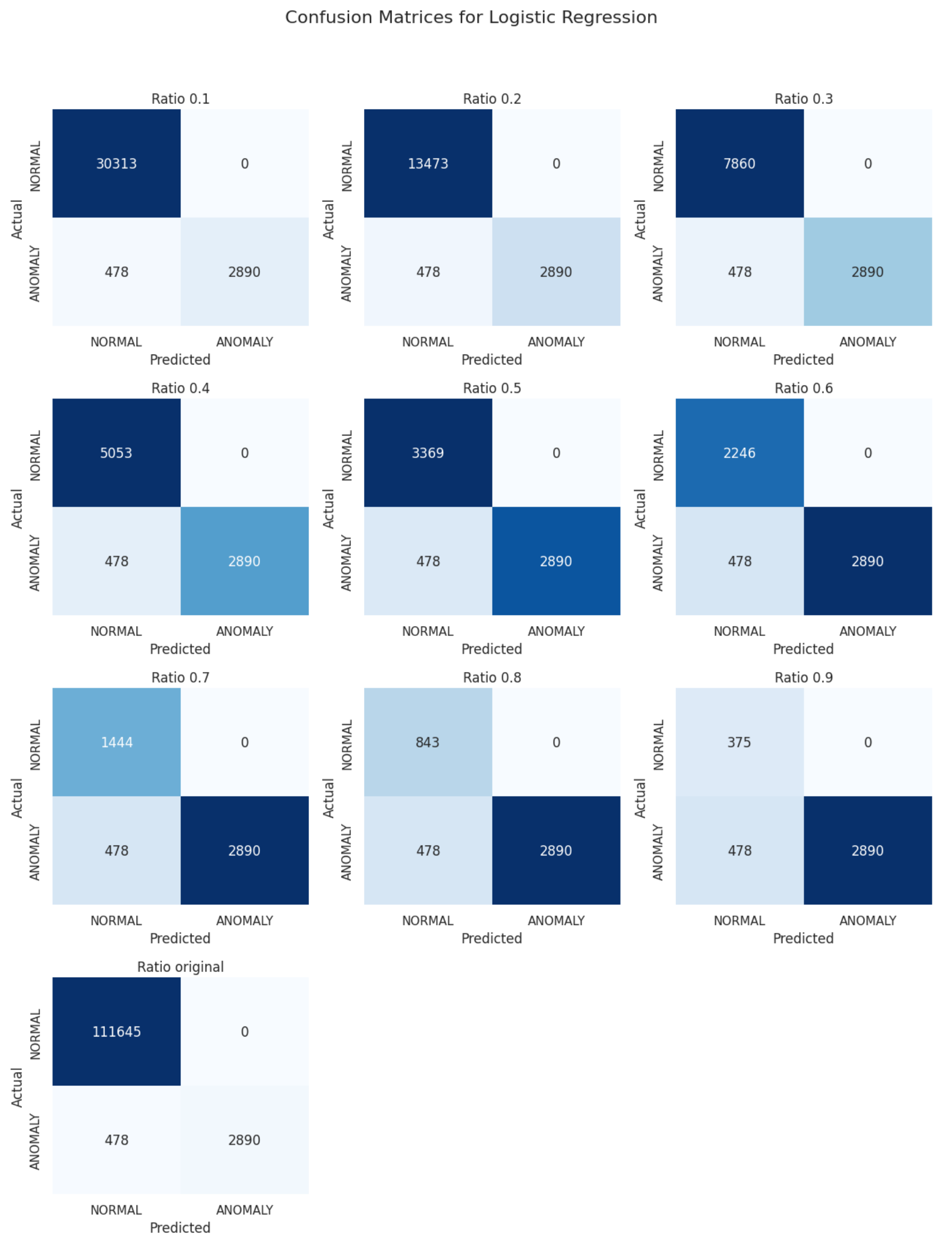

Figure 2: Confusion Matrix for Logistic Regression on different log to data ratio.

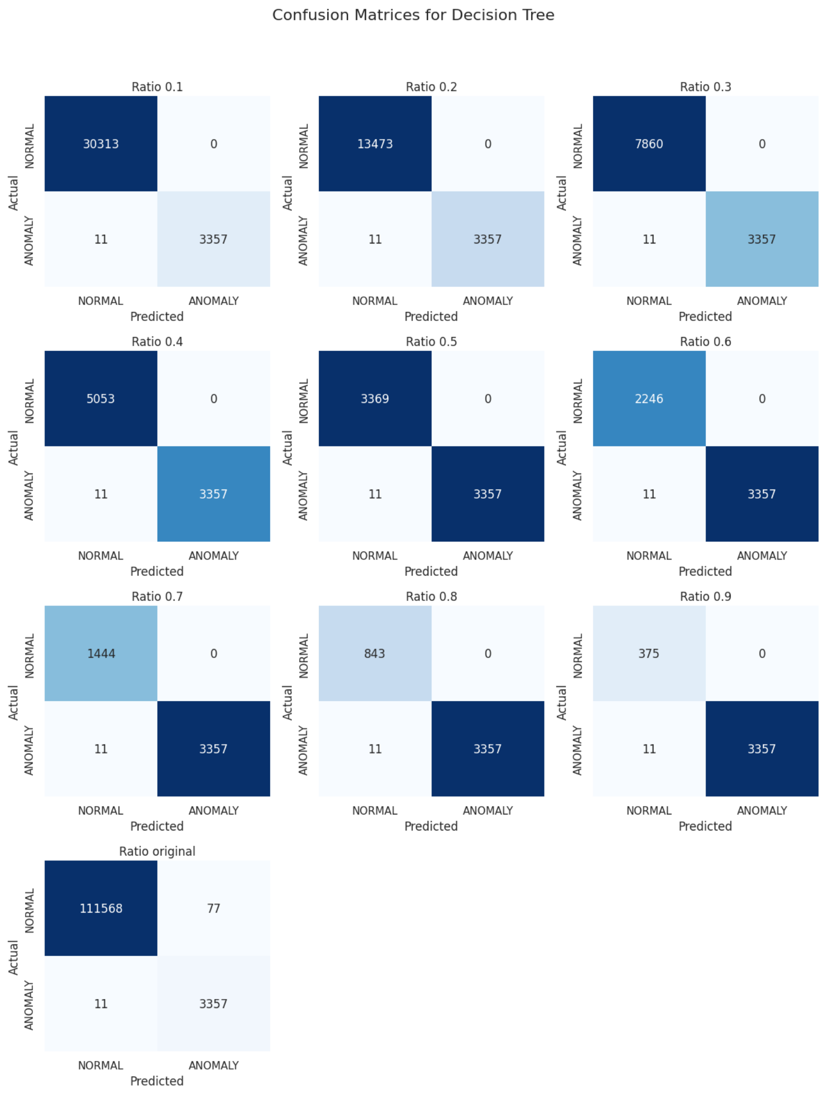

Figure 3: Confusion matrix for Decision Tree on different log to data ratio.

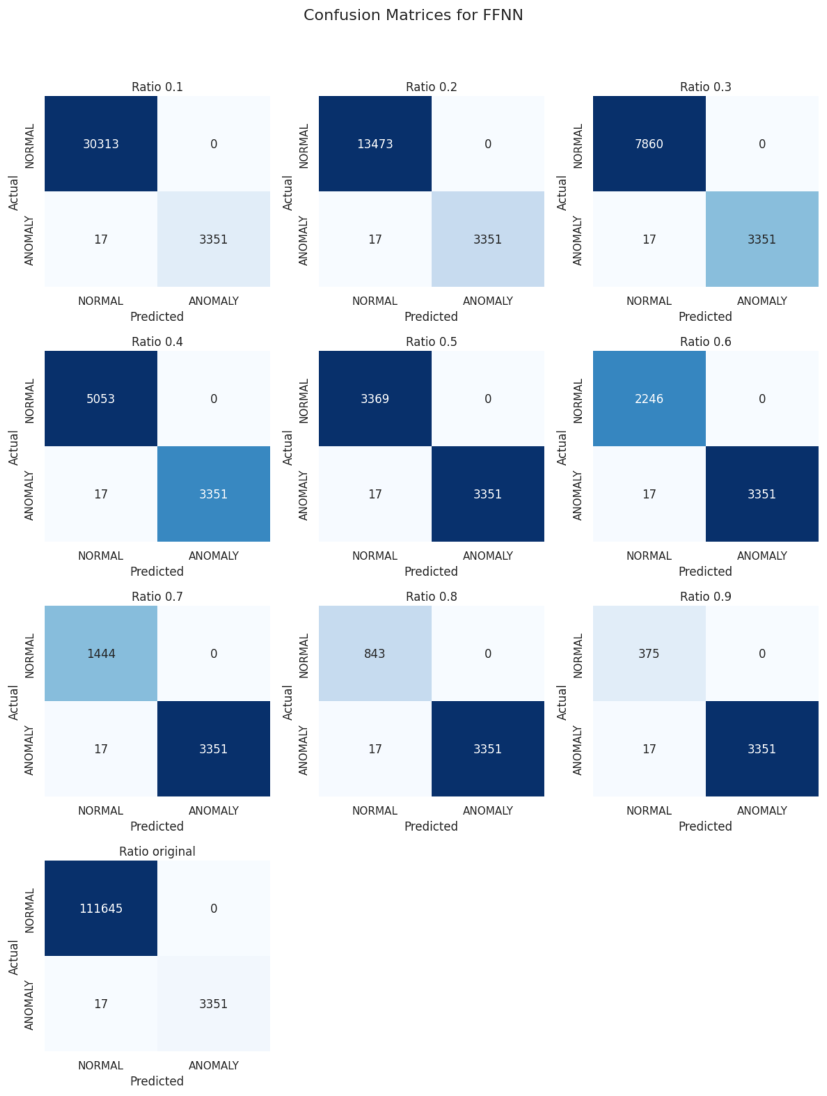

Figure 4: Confusion matrix for FFNN on different log to data ratio.

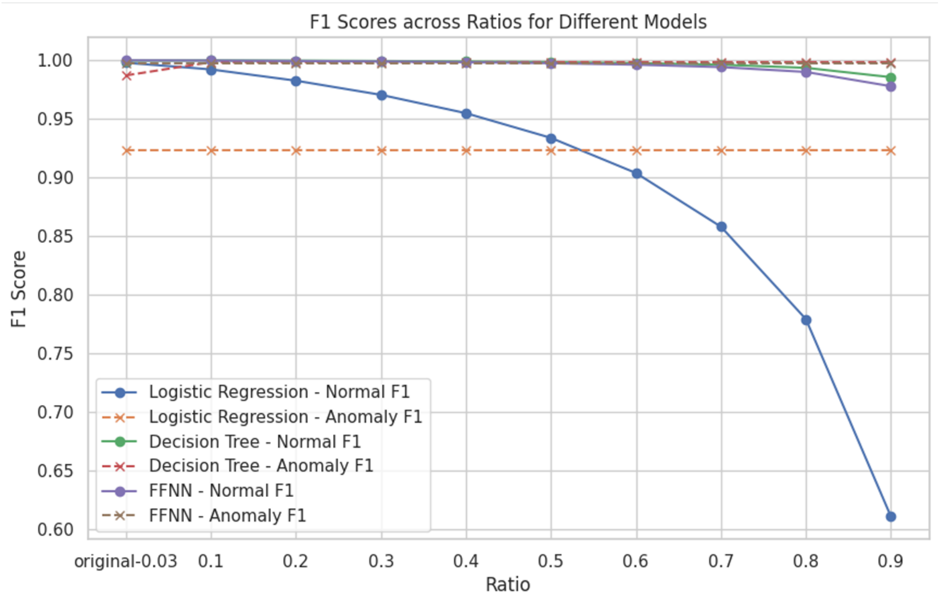

Figure 5: F1 score across ratio for different models

The graph 5 displays the F1 scores for various machine learning models (Logistic Regression, Decision Tree, and Feedforward Neural Network (FFNN)) across different ratios of normal and anomaly data. The F1 scores are presented separately for normal and anomaly classes, which helps to evaluate the models' performance on both types of data. The Logistic Regression model's F1 score for normal data shows a declining trend as the ratio approaches 0.9, indicating a significant drop in performance, whereas its anomaly F1 score remains consistent. Both the Decision Tree and FFNN models maintain high F1 scores across all ratios for both normal and anomaly classes, demonstrating robustness to changes in the data ratio. Overall, the Decision Tree and FFNN appear to perform consistently better across all ratios, whereas Logistic Regression struggles significantly, especially for the normal data as the data ratio varies. This suggests that Decision Tree and FFNN models are better suited for handling both normal and anomaly data in different data distributions.

5.1 Frontend Dashboard

The image shows the visual representation of log data collected from Kubernetes pods in real-time. This dashboard presents the error and non-error counts for a selected namespace and container over a specified time range. Users can easily select a namespace, container, and time range to filter the displayed data, which helps monitor and understand the system's behavior. The graph shown in the dashboard enables users to observe the distribution of log entries, highlighting potential anomalies or trends in the container's performance through stacked bar charts of error and non-error counts. This interface offers an intuitive way to visualize and track log metrics, supporting proactive system management.

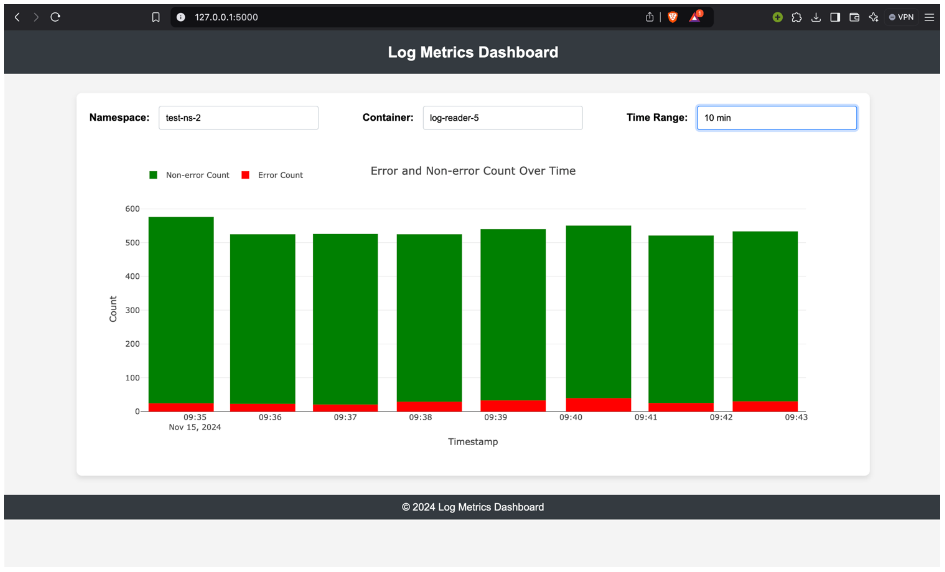

Figure 6: Sample Output from Frontend

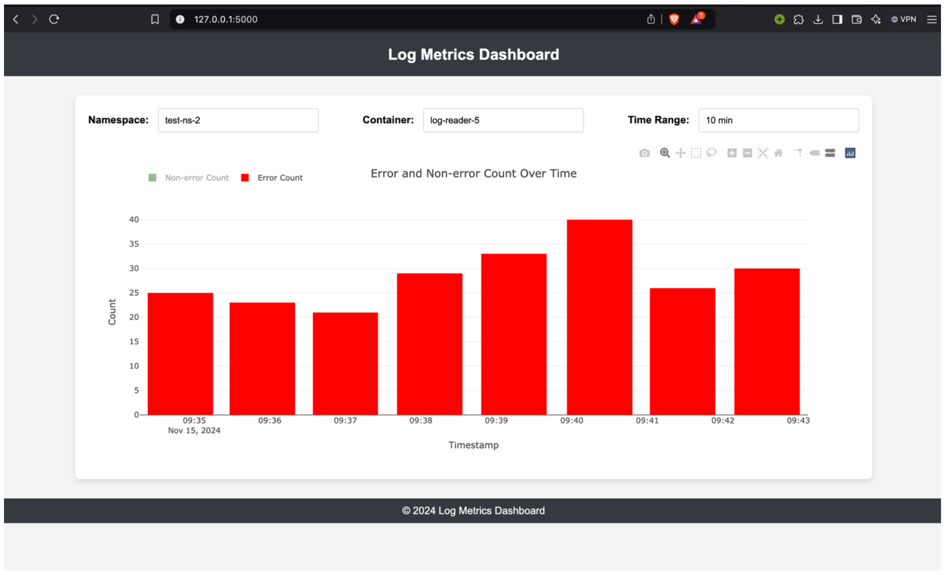

Figure 7: Error counts overtime

6 Conclusions and Future Directions

In conclusion, the Logminder project successfully showcases the potential of automated log analysis for managing complex systems through advanced data processing and anomaly detection. Using preprocessing techniques like template-based event mapping, TF-IDF, and GloVe embeddings, we transformed unstructured log data into coherent vector representations, enhancing anomaly detection accuracy. Our method ensured that temporal and contextual structures were preserved, and with Docker and Kubernetes, we established a scalable, resilient setup for real-time log monitoring across distributed environments.

Looking ahead, a key area for future development is expanding the system's applicability beyond the HDFS dataset, making it compatible with multiple file system types. To achieve this, a more generalized log processing pipeline could be created to adapt to the diverse formats and structures of logs from different file systems, such as NTFS, ext4, and APFS. This enhancement would involve implementing flexible data extraction and parsing methods capable of interpreting and standardizing logs regardless of their origin. Such a system would provide a unified anomaly detection framework that is versatile across varied storage systems, allowing organizations to monitor, analyze, and detect issues in a wide range of infrastructure setups without the need for file system-specific adjustments. Additionally, we aim to expand the model's applicability to datasets where fully labeled data and predefined log templates are not available. To address the challenge of limited labeled data, a method similar to the Probabilistic Label Estimation technique described in this paper[3] could be employed. Moreover, approaches could be developed to extract log templates directly from raw log files during training, ensuring that these extracted templates can be effectively utilized during live testing.

7 Team Contributions

7.1 Ritam Acharya

- Literature Review.
- Documentation.
- ML model Decision Tree.
- Frontend.

7.2 Jatin Jangir

- Ideation.
- Database integration.
- Docker and Kubernetes integration.
- Data preprocessing.
- Application architecture design.
- Literature review.

7.3 Milan Roy

- Literature review.
- Implementation of a paper.
- Solution of challenges faced in the implementation.

- Methodology and implementation of data preprocessing.
- Creation of test set to be used in live setting.
- ML model architecture and training.
- Model comparisons.

7.4 Ananya Chaturvedi

- Ideation.
- Literature review.
- Implemented test data of different log ratios on different ML and DL models.
- Data Preprocessing.
- Model training.

References

[1] Docker: Lightweight Linux Containers for Consistent Development and Deployment. Available at `https://www.researchgate.net/publication/261960832_Docker_lightweight_Linux_containers_for_consistent_development_and_deployment`.

[2] Kubernetes: Automating Deployment, Scaling, and Management of Containerized Applications. Available at `https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/44843.pdf`.

[3] L. Yang et al., ”Semi-Supervised Log-Based Anomaly Detection via Probabilistic Label Estimation,” 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), Madrid, ES, 2021, pp. 1448-1460, doi: 10.1109/ICSE43902.2021.00130.

[4] Drain Parser: A log parser used to generate log templates from raw log files. Available at `https://github.com/logpai/logparser/tree/main/logparser/Drain`

[5] Le, V., Zhang, H. (2022). Log-based Anomaly Detection with Deep Learning: How Far Are We? ArXiv. https://doi.org/10.1145/3510003.3510155