

LOG-BASED ANOMALY DETECTION USING PRE-TRAINED LARGE LANGUAGE MODELS

by

Smit Shah
AU1920088

A thesis submitted in partial fulfillment
of the requirements for the My Degree
degree in My Program Name in the
Undergraduate Studies of
Ahmedabad University

March 2023

Thesis Proposal/Approval Form

Date: April 28, 2023

This undergraduate thesis has been examined and approved.

Review Committee:

Signature

Professor
Prof. Srikrishnan Divakaran, March 2023

Signature

Professor
External Examiner's Name, March 2023

Approved:

Signature

Professor Raghavan Rangarajan
Dean of the School of Arts and Sciences, March 2023

Signature

Professor Darshini Mahadevia or Professor Balaji Prakash
Associate Dean of the School of Arts and Sciences, March 2023

Copyright by
Smit Shah: Smit Shah
2023: 2023
All Rights Reserved

ACKNOWLEDGMENTS

I would like to express my gratitude to Prof. Srikrishnan Divakaran for his guidance and support during my undergraduate thesis. His expertise and encouragement were crucial in developing my understanding of the subject and producing a quality research project. I am thankful for his dedication and patience.

I would also like to thank my parents for their unwavering love and support. They have been my source of strength and inspiration throughout my academic journey. Their faith in me and constant encouragement have kept me motivated and focused. I am truly blessed to have them in my life.

ABSTRACT

Anomaly detection in log data refers to the process of identifying patterns or events in log data that deviate from the expected behavior or normal patterns. This is important for detecting potential issues in systems and ensuring that they are operating efficiently and securely. The evolution of techniques used for anomaly detection in log data has seen significant advancements over the years. Initially, rule-based systems were used to identify anomalies, but these systems were often limited by the specificity of the rules and were unable to adapt to changes in the data over a period of time. With the advent of machine learning and artificial intelligence, new techniques were developed that leveraged these technologies to improve the accuracy and scalability of anomaly detection.

Machine learning techniques, such as clustering, decision trees, and neural networks, have been widely used for anomaly detection in log data. In recent years, the use of Long Short-Term Memory (LSTMs) and transformers has become popular for this purpose. LSTMs are a type of recurrent neural network that are well-suited for handling time-series data and have shown great performance in detecting anomalies in log data. While the LSTM was a significant breakthrough in machine learning by enabling models to retain memory, the Transformer architecture goes further by introducing the self-attention mechanism and the ability to process entire sequences of input data in parallel. Transformers are a type of deep learning model that have been applied to a wide range of natural language processing (NLP) tasks and are now being used in anomaly detection in log data to analyze and identify patterns in text data.

This research focuses on using pre-trained language models, such as BERT and GPT, for anomaly detection in server logs. The study aims to evaluate the effectiveness of these models in detecting abnormal patterns and to identify suitable method for log parsing in this context. The research also aims to understand the explainability of the models and their limitations to identify areas for improvement and provide a comprehensive understanding of their performance. The study will use log datasets from loghub, a publicly available platform, to ensure that the results are generalizable and aligned with industry standards. Overall, this research aims to contribute to de-

veloping more effective methods for log-based anomaly detection, which is crucial for identifying potential security breaches, system failures, or other events that may require attention.

TABLE OF CONTENTS

LIST OF TABLES	vi
LIST OF FIGURES	vii
INTRODUCTION	1
LITERATURE REVIEW	5
RESEARCH QUESTIONS	16
EXPERIMENTAL SETUP AND STUDY	18
Tools	18
OpenAI API	18
Text-Davinci-003	18
Datasets	18
Apache	18
OpenSSH	19
Automated log parsing	20
Challenges	24
REFERENCES	28
APPENDIX A: CODE AND IMAGES	30
Code	30

LIST OF TABLES

Table 1. Comparing log parsers	23
--	----

LIST OF FIGURES

Figure 1. LSTM By Dobilas (2022).	5
Figure 2. Deeplog Architecture By Du, Li, Zheng, and Srikumar (2017).	6
Figure 3. Transformer Architecture By Vaswani, Shazeer, Parmar, et al. (2017).	7
Figure 4. Bidirectional Encoder Representation from Transformers By Halthor (2020) . . .	9
Figure 5. LogBERT By Halthor (2020)	9
Figure 6. Log Anomaly Detection Framework Chen, Liu, Gu, et al. (2021)	11
Figure 7. Apache Web Server Logs	19
Figure 8. OpenSSH Server Logs	20
Figure 9. Log Parsing	21
Figure 10. Original Log File	22
Figure 11. Parsed Logs	22
Figure 12. Loading Ground Truth File into the data frame	22
Figure 13. Log Parsing	22
Figure 14. Log Parsing	22
Figure 15. Comparision with 13 other parsers	24
Figure 16. Unnecessary Inference 'user=root'	25
Figure 17. Deviation in log format	26
Figure 18. Deviation in log format	26
Figure 19. SSH Log Parsing	30

INTRODUCTION

Anomaly detection is identifying patterns, events, or observations in data that deviate significantly from the norm. It is used to identify unusual or abnormal behavior in a dataset. The need for anomaly detection is to detect unusual patterns, which can indicate a problem or potential threat. For example, in cybersecurity, anomaly detection can be used to detect intrusions in network traffic; in manufacturing, it can be used to detect equipment failures; and in finance, it can be used to detect fraud. In general, anomaly detection is helpful in any situation where it is vital to identify deviations from normal behavior in order to detect potential issues or problems.

Log-based anomaly detection is a method for identifying abnormal or unusual patterns in log data. This can be useful for detecting potential security breaches, system failures, or other events that may require attention. The technique involves analysing log data and identifying patterns that deviate significantly from normal behavior. Various algorithms and approaches can be used to detect anomalies, including machine learning algorithms and statistical methods. One of the main challenges of log-based anomaly detection is dealing with the fuzziness in data characteristics. Fuzziness refers to the imprecision and uncertainty that can exist in the data due to factors such as noise, missing data, outliers, and variations in data patterns over time. In log-based anomaly detection, the data being analyzed typically consists of large, complex datasets with a high degree of variability and nonlinearity. This can make it difficult to identify standard patterns and accurately differentiate them from abnormal patterns.

In the early days, anomaly detection in log data was mainly performed using simple statistical techniques such as mean, median and standard deviation. However, as the amount of log data increased, these methods became insufficient and new techniques had to be developed to handle the volume and complexity of the data.

One of the early advancements in log-based anomaly detection was the use of machine learning techniques, such as clustering, decision trees, and neural networks. Clustering algorithms group similar data points together, while decision trees and neural networks are used to build

predictive models to identify anomalies. These methods provided improved accuracy in detecting anomalies in log data, but were still limited by their inability to handle complex data patterns varying over time.

The use of Long Short-Term Memory (LSTMs) and transformers has revolutionized log-based anomaly detection. LSTMs are a type of recurrent neural network that are designed to handle sequential data, making them well-suited for log-based anomaly detection. LSTMs can analyze long sequences of log data and identify patterns in the data that may indicate an anomaly.

While the LSTM was a significant breakthrough in machine learning by enabling models to retain memory, the Transformer architecture goes further by introducing the self-attention mechanism and the ability to process entire sequences of input data in parallel. This allows Transformers to capture complex relationships between elements in the sequence and access all elements in the input sequence for making predictions, leading to state-of-the-art performance on various NLP tasks.

The self-attention mechanism allows the Transformer to process entire sequences of input data in one go, rather than processing them one element at a time. This enables the model to capture relationships between different elements in the sequence and identify which elements are most relevant for predicting the output. The infinite reference window allows the Transformer to access all elements in the input sequence when making predictions, regardless of their position in the sequence. This is in contrast to earlier sequence models, such as RNNs and LSTMs, which have a limited "memory" of past inputs due to the nature of their sequential processing.

Together, these innovations in the Transformer architecture enable it to achieve state-of-the-art performance on a wide range of NLP tasks, including language modeling, machine translation, and question answering.

In log-based anomaly detection, transformers are used to analyze text data in log files and identify patterns in the data that may indicate an anomaly. Transformers have the advantage of being able to handle a large amount of text data, and can also learn from the context of the data to provide more accurate results.

Pre-trained, and large language models were made possible by the breakthrough made by transformer-based architectures, which were introduced in 2017 with the paper "Attention is All You Need" by Vaswani et al.

The transformer architecture significantly improved over previous NLP models because it used self-attention to allow the model to attend to all positions in an input sequence simultaneously. This was a departure from earlier models, which used recurrent or convolutional neural networks to process sequences one element at a time. The transformer architecture allowed for more efficient processing of long sequences and better modeling of relationships between different elements in the sequence.

In 2018, researchers at Google introduced the Bidirectional Encoder Representations from Transformers (BERT) model, which was pre-trained on massive amounts of text data using a transformer-based architecture. BERT achieved state-of-the-art performance on a wide range of NLP tasks by fine-tuning the pre-trained model on specific tasks.

Following the success of BERT, other large language models such as GPT-2 and T5 were introduced, which were also based on transformer architectures and pre-trained on large amounts of data. These models have continued to push the boundaries of NLP performance, and their success has led to further research into even larger and more powerful language models.

The potential application of pre-trained large language models in log-based anomaly detection is significant. These models can be used to analyze log data in real time, identify patterns in the text data, and detect anomalies that may indicate system problems or security threats. The ability of pre-trained large language models to handle large amounts of text data, understand the context of the data, and identify patterns in the data can provide more accurate results compared to traditional log-based anomaly detection methods.

For example, pre-trained large language models can be fine-tuned to identify specific log data patterns that are indicative of a particular system problem or security threat. These models can then be used to monitor log data in real time and detect anomalies as they occur. The use of pre-trained large language models in log-based anomaly detection can also help reduce the time

required to develop custom models and provide a more scalable solution for monitoring log data.

In conclusion, pre-trained large language models have the potential to enhance the accuracy and efficiency of log-based anomaly detection greatly. The ability of these models to handle large amounts of text data, understand the context of the data, and identify patterns in the data provides a significant advantage over traditional log-based anomaly detection methods.

This research focuses on the methods of feature extraction for anomaly detection in server logs, and the impact of these methods. The study uses pre-trained language models such as BERT and GPT to extract features from server logs. The goal is to evaluate the effectiveness of these pre-trained models in detecting anomalies and to identify the most suitable method for feature extraction in this context. The study aims to provide insights into the use of pre-trained language models for anomaly detection in server logs and to contribute to the development of more effective methods for this task.

The research also aims to understand the explainability of the used models and their limitations. This will help identify the models' strengths and weaknesses and identify areas for improvement. Additionally, the study will investigate the limitations of the models, such as their ability to handle complex and diverse server logs. Overall, this research will provide a comprehensive understanding of the performance and limitations of pre-trained language models for anomaly detection in server logs.

LITERATURE REVIEW

Anomaly detection is the process of identifying unusual patterns in data, which is used to detect potential issues or problems. Log-based anomaly detection is a method for identifying abnormal or unusual patterns in log data, which is particularly useful for cybersecurity, manufacturing, and finance. Various algorithms and approaches, including machine learning algorithms and statistical methods, can be used to detect anomalies. However, one of the main challenges of log-based anomaly detection is dealing with the fuzziness in data characteristics due to noise, missing data, outliers, and variations in data patterns over time. In the early days, simple statistical techniques were used, but with increasing log data, machine learning techniques such as clustering, decision trees, and neural networks were developed. These methods provided improved accuracy but were still limited by their inability to handle complex data patterns varying over time.

LONG SHORT-TERM MEMORY NEURAL NETWORKS

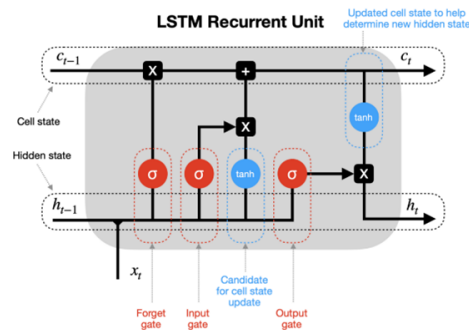


Figure 1. LSTM By Dobilas (2022).

LSTM, or Long Short-Term Memory, is a type of recurrent neural network (RNN) that is particularly well-suited for modeling sequential data. One application of LSTM is in log anomaly detection, where the goal is to detect anomalous behavior in system logs that may indicate security breaches, system failures, or other types of problems.

In log anomaly detection, the challenge is to distinguish between normal log messages and

anomalous messages, which may be infrequent and difficult to predict. LSTMs are well-suited for this task because they can learn to model the complex relationships between different log messages and identify patterns that may be indicative of anomalous behavior.

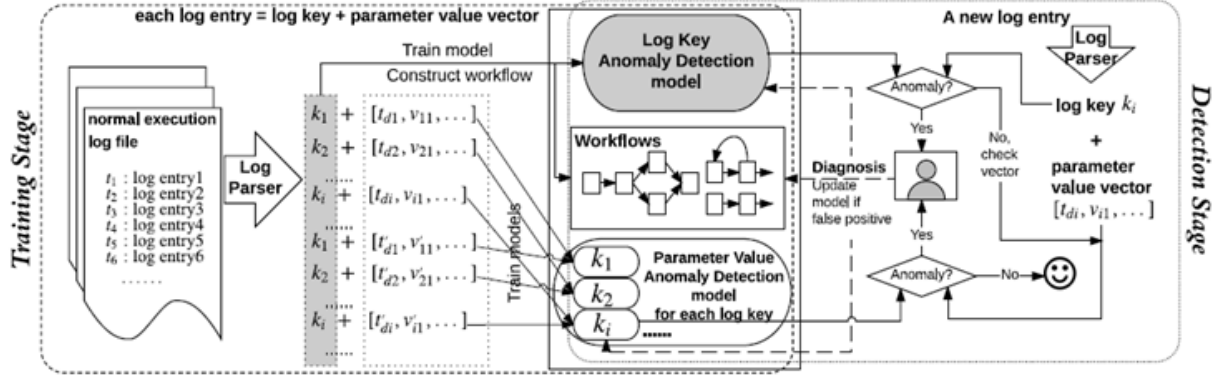


Figure 1: DeepLog architecture.

Figure 2. Deeplog Architecture By Du, Li, Zheng, and Srikumar (2017).

One notable implementation of LSTM for log anomaly detection is DeepLog, introduced by Fei Fei Lee and colleagues in 2017. DeepLog uses an LSTM-based architecture to learn a representation of system logs and identify anomalous behavior. It used LSTM to model system logs as natural language sequences. This innovative approach made it more effective than other existing methods because it automatically learned log patterns from normal execution and detected anomalies when log patterns deviated from the model trained from log data under normal execution.

One significant advantage of DeepLog is its ability to adapt to new log patterns over time and construct workflows from the underlying system log to help users diagnose problems. Another notable feature is its mechanism for user feedback, which allows it to adjust its weights dynamically online over time to adapt itself to new system execution patterns.

Compared to other generic methods (statistical/traditional ML) for anomaly detection, DeepLog has several advantages. It can be used for online anomaly detection in a streaming fashion with high accuracy, even when trained on only a small fraction of log entries corresponding to normal system execution. It can also build workflow models from log entries to help users diagnose a problem once an anomaly is identified. According to Du, Li, Zheng, and Srikumar (2017), the “HDFS log dataset explored by previous work [22, 39], trained on only a very small fraction (less

than 1%) of log entries corresponding to normal system execution, DeepLog can achieve almost 100% detection accuracy on the remaining 99% of log entries.”

Moreover, DeepLog does not require any domain-specific knowledge and can use timestamps and parameter values for anomaly detection, which were previously missing in other methods. Overall, DeepLog is a powerful and versatile tool that can help businesses detect and diagnose anomalies in their systems quickly and effectively.

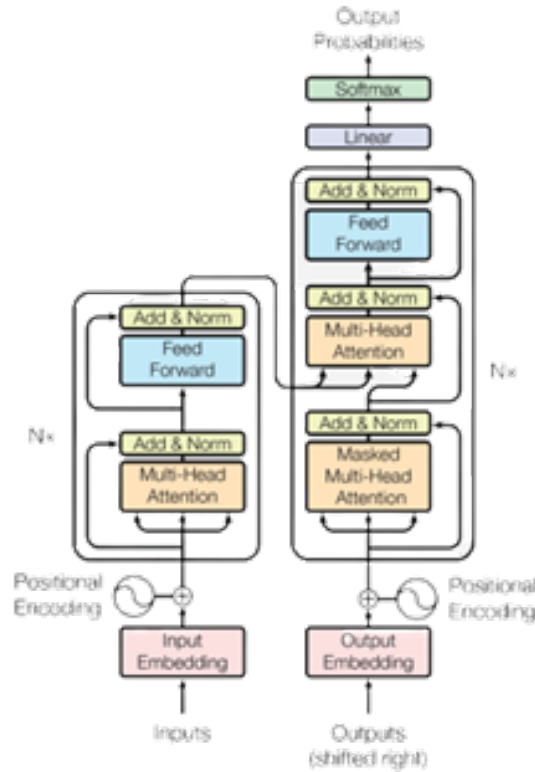


Figure 3. Transformer Architecture By Vaswani, Shazeer, Parmar, et al. (2017).

Although the LSTM was a significant milestone in machine learning as it allowed models to store information, the Transformer architecture took it to the next level by incorporating self-attention mechanism and the capability to process entire input sequences in parallel. The Transformer architecture was introduced in 2017 by Vaswani et al. in their paper "Attention Is All You Need." This paper presented a novel architecture for sequence modeling and introduced the self-attention mechanism and the ability to process entire sequences of input data in parallel.

The Transformer quickly gained attention and became a popular architecture in natural language processing (NLP) due to its superior performance on various tasks such as language modeling, machine translation, and text generation. Since its introduction, the Transformer has been further developed and modified, and it remains a popular architecture for NLP models to this day.

One of the limitations of the LSTM is that it processes input sequences sequentially, which makes it difficult to capture relationships between elements that are far apart in the sequence. In addition, the LSTM's memory is finite, so it can only retain information from a limited number of past elements in the sequence.

On the other hand, the Transformer architecture overcomes these limitations by introducing the self-attention mechanism and the ability to process entire sequences of input data in parallel. The self-attention mechanism allows the model to attend to all elements in the input sequence when making predictions, regardless of their position in the sequence, and the parallel processing allows it to capture relationships between all elements in the sequence, even if they are far apart.

Furthermore, the Transformer's ability to process entire sequences in parallel makes it much faster than LSTMs, especially for long sequences. Additionally, the Transformer does not suffer from the vanishing gradient problem that LSTMs can encounter, which can make training difficult for very deep networks.

BERT (Bidirectional Encoder Representations from Transformers) was developed using the Transformer architecture. It builds on the Transformer architecture by adding a novel pre-training objective. Specifically, BERT is pre-trained on a large corpus of text using a masked language modeling (MLM) task and a next sentence prediction (NSP) task. The MLM task involves randomly masking out words in a sentence and training the model to predict the missing words based on the context. The NSP task involves predicting whether two sentences are consecutive in the text or not.

The pre-training step allows BERT to learn rich representations of language that capture both local and global context. The model can then be fine-tuned on a wide range of downstream NLP tasks, such as text classification, question answering, and named entity recognition.

Bidirectional Encoder Representation from Transformers

Pretraining (Pass 1) : “What is language? What is context?”

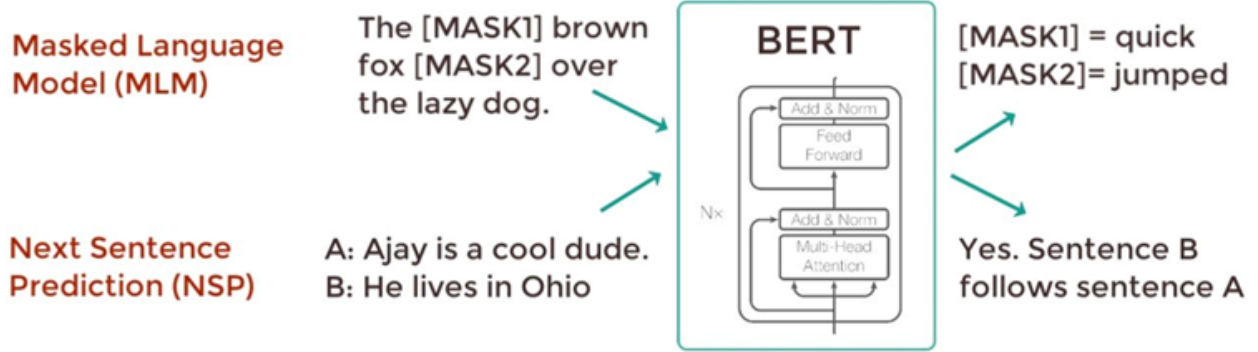


Figure 4. Bidirectional Encoder Representation from Transformers By Halthor (2020)

BERT was a major breakthrough in NLP because it achieved state-of-the-art performance on a wide range of tasks with relatively little task-specific training data. It also paved the way for other large-scale pre-trained language models, such as GPT and XLNet.

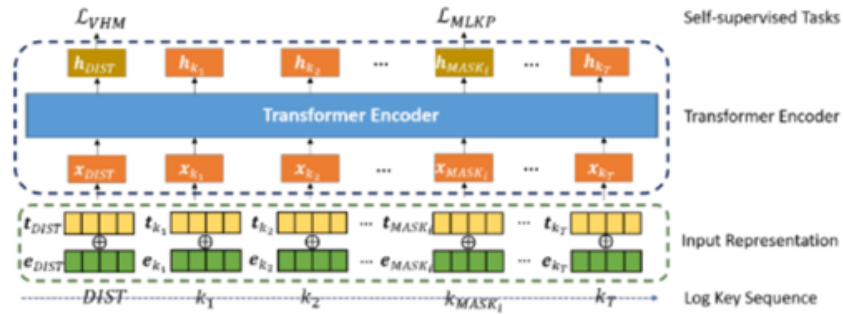


Figure 5. LogBERT By Halthor (2020)

LogBERT was introduced by Haixuan Guo, Shuhan Yuan, and Xintao Wu in 2021 to address the limitations of using RNN-based models for log anomaly detection. RNNs are widely used for modeling sequential data, including log data, but they have certain limitations. Specifically, RNNs cannot make each log in a sequence encode the context information from both the left and right context, which is crucial for detecting malicious attacks based on log messages. Moreover, current RNN-based anomaly detection models are trained to capture the patterns of normal

sequences by predicting the next log message given previous log messages. However, this training objective mainly focuses on capturing the correlation among the log messages in normal sequences and cannot explicitly encode the common patterns shared by all normal sequences.

To overcome these limitations, LogBERT is proposed as a self-supervised framework for log anomaly detection based on Bidirectional Encoder Representations from Transformers (BERT). LogBERT leverages BERT to capture patterns of normal log sequences, and its structure allows for the contextual embedding of each log entry to capture the information of the entire log sequences. Two self-supervised training tasks are proposed: masked log key prediction and volume of hypersphere minimization. These tasks aim to correctly predict log keys in normal log sequences that are randomly masked and make the normal log sequences close to each other in the embedding space, respectively. After training, LogBERT encodes the information about normal log sequences, and a criterion is derived to detect anomalous log sequences based on LogBERT. Experimental results on three log datasets (HDFS, BGL and Thunderbird Mini) show that LogBERT achieves the best performance on log anomaly detection compared to various state-of-the-art baselines.

LogBERT uses two self-supervised training tasks to capture the contextual information of log sequences. The first task is called masked log key prediction, which aims to correctly predict log keys in normal log sequences that are randomly masked. The second task is called the volume of hypersphere minimization, which aims to make the normal log sequences close to each other in the embedding space. It also uses the Transformer encoder to learn the contextual relations among log keys in a sequence, and the input representation of each log key is defined as the combination of the log key embedding matrix and the position embeddings generated by a sinusoid function. The performance metrics used to compare LogBERT to existing approaches are precision, recall, and F1 score.

The criteria used to identify anomalous log sequences based on LogBERT is derived from the prediction results on the MASK tokens. It also uses the prediction accuracy of the masked log keys as an indicator of whether a log sequence is normal or anomalous. It also uses a spherical objective function to make the normal log sequences close to each other in the embedding space

and to make the anomalous log sequences have a larger distance to the center.

The main advantages of LogBERT over traditional machine learning models are its ability to leverage the information from other log sequences via the center representation c , its ability to predict the masked log keys with higher accuracy for normal log sequences, and its ability to detect anomalous log sequences after being trained on normal log sequences. Furthermore, LogBERT can capture the patterns of log sequences more effectively than LSTM models, and combining two self-supervised tasks to train LogBERT results in better performance than models only trained by one task, especially when log sequences are relatively short. It is able to capture the temporal information of discrete log messages, is able to model the sequential data, and is able to detect malicious attacks based on a sequence of operations where each log entry looks normal. Additionally, LogBERT leverages the information from other log sequences via the center representation and is able to detect anomalous log sequences by deriving a criterion based on LogBERT after training.

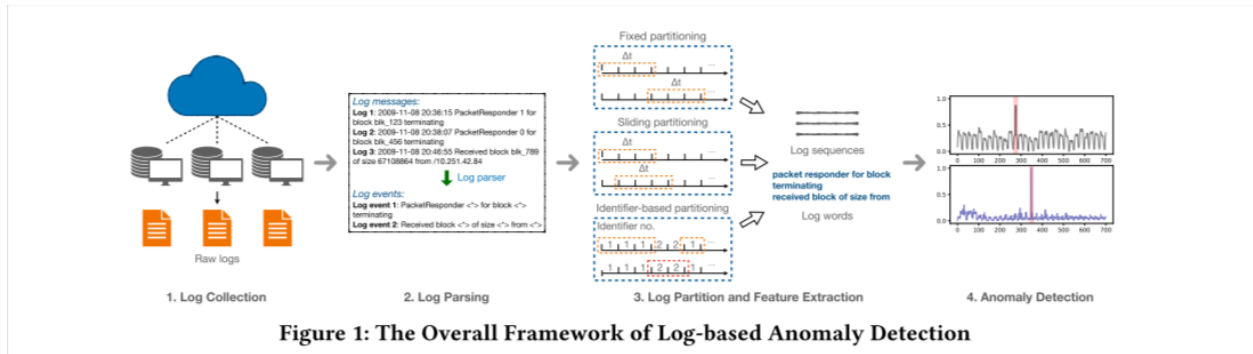


Figure 1: The Overall Framework of Log-based Anomaly Detection

Figure 6. Log Anomaly Detection Framework Chen, Liu, Gu, et al. (2021)

According to Chen, Liu, Gu, et al. (2021) the log anomaly detection framework is a four-phase process that involves log collection, log parsing, feature extraction, and anomaly detection. In the log collection phase, software systems print logs to the system console or designated log files to record runtime status. These logs are collected and analyzed for tasks such as anomaly detection and fault localization. However, the large volume of collected logs can overwhelm existing troubleshooting systems, and the lack of labeled data poses difficulty in analysis.

In the log parsing phase, the raw logs are often semi-structured and need to be parsed into

a structured format for downstream analysis. The constant/static part and variable/dynamic part of a raw log line are identified. The constant part is commonly referred to as log event, log template, or log key, while the variable part stores the value of the corresponding parameters.

In the log partition and feature extraction phase, each log message is represented with the log template identified by a log parser, and log timestamp and log identifier are often employed to partition logs into different groups, each representing a log sequence. The log sequences are then converted into numerical features that can be understood by machine learning/deep learning algorithms. Traditional machine learning-based methods generate a vector of log event count as the input feature, while deep learning-based methods directly consume the log event sequence.

Finally, in the anomaly detection phase, the log features constructed in the previous phase are used to detect anomalies. Various machine learning and deep learning-based methods can be used for this purpose. The goal is to learn the semantics of logs to make more intelligent decisions. The log-based anomaly detection framework can be used in various applications such as cybersecurity, system performance monitoring, and fault diagnosis.

The collection of log datasets is crucial for software development and maintenance. Logs contain valuable system runtime information that is essential for understanding the behavior of software systems. With the rapid growth of software size and complexity, there is a pressing need to efficiently and effectively handle large volumes of logs.

Logs contain rich information that is essential for various log-based system management tasks, such as anomaly detection, duplicate issue identification, usage statistics analysis, and program verification. However, as software systems continue to grow in scale and complexity, the volume and velocity of system logs also increase rapidly, reaching up to 50 GB/hour.

To deal with data velocity and complexity, researchers have been exploring the use of AI techniques for intelligent log analytics. However, the success of these techniques depends heavily on the availability of public log datasets and appropriate benchmarking methods. Unfortunately, only a small fraction of these techniques have been successfully deployed in the industry due to the lack of standard datasets.

By collecting log datasets and benchmarking them, S. He, Zhu, He, and Lyu (2020) facilitates research in the field of intelligent log analytics. It enables the development of more effective and efficient techniques that can be successfully deployed in the industry. Moreover, the availability of these datasets allows researchers to train and experiment with AI models, leading to further advancements in the field.

To facilitate research in log analytics, a collection of system logs is maintained by Loghub, which is freely accessible for research purposes. These logs consist of production data released from previous studies and logs collected from real systems in the lab environment. The logs are not sanitized, anonymized, or modified to maintain their integrity. The collection comprises over 77 GB of logs, which can be challenging to manage

For the convenience of researchers, a small sample of each dataset is hosted on the project website, while the complete versions can be obtained by requesting through Zenodo, an open dataset sharing website. LogHub’s collection of system logs comprises datasets from various types of systems, including distributed systems, operating systems, supercomputer systems, mobile systems, and server applications. This diverse collection provides researchers with a broad range of data to analyze and test their log analytics techniques. The inclusion of these datasets also enables researchers to explore the differences in log data across various types of systems, which can aid in the development of more specialized log analytics tools.

Log parsing or feature extraction is a critical step in AI-powered log analysis for detecting anomalies accurately and quickly. System logs are often unstructured texts, containing several fields and natural language descriptions written by developers, making it challenging for log analysis algorithms to work with them. Log parsing helps to transform unstructured log messages into structured system events, enabling log analysis algorithms to detect anomalies effectively.

Traditional parsing approaches that rely on manual parsing rules construction become inefficient and labor-intensive, especially with the rapid increase in log volumes. To address this problem, recent research has proposed various data-driven log parsers that can automatically label an unstructured log message with a corresponding system event ID. Data-driven log parsers use

similarity metrics to separate unstructured log messages into different clusters accurately and efficiently. This allows log parsers to summarize corresponding system events and match each log message with an event ID. The structured logs, i.e., log messages with event IDs, can be easily transformed into a matrix or utilized directly by log analysis algorithms.

According to S. He, Zhu, He, and Lyu (2020) the impact of better log parsing is significant in terms of improving downstream tasks such as anomaly detection and failure identification. The text suggests that a semantic parser was used to derive rich semantics from log messages collected from widely-applied systems. With an average F1 score of 0.985, the effectiveness of the semantic parser is demonstrated.

The downstream models show an improvement in performance with appropriately extracted semantics. Anomaly detection datasets show improvement by 1.2%-11.7%, while failure identification datasets improve by 8.65%. This improvement indicates that better log parsing can lead to more accurate identification of anomalies and failures, which can help in improving system performance and preventing downtime. Therefore, the impact of better log parsing is significant in enhancing the overall efficiency and reliability of complex systems.

According to Guo, Yuan, and Wu (2021), most existing log parsers can be categorized into one of the following categories: Clustering, Frequent Pattern Mining, Heuristic approaches, and Program Analysis. The authors also noted the rapid growth in the number of parsers proposed during the span of four-year intervals between 2002-2021. Despite the availability of various methods to parse logs choosing and configuring a suitable log parser from the extensive range available can be a challenging task. The reason being, existing log parsers are implemented using various algorithms, which result in significant variance in their performance features and operational complexities.

The performance and configuration options across the available parsers differ greatly. The incorrect configuration of a parser can lead to severe performance degradation, resulting in complications in log mining and analytic tasks. Therefore, careful consideration of the specific requirements of the use case is essential in selecting a log parser with optimal performance features and

configuration options to avoid potential misconfiguration issues. As a result of the aforementioned factors data representation is one of the major challenges in log-based anomaly detection; log data is often heterogeneous and includes unstructured messages and categorical parameters, making it difficult to feed into neural networks that typically require structured and numeric input data.

Additionally as mentioned by Landauer, Onder, Skopik, and Wurzenberger (2022), anomalies can manifest in various ways, making it challenging to design detection techniques that are applicable to all types of anomalies. This is further complicated by the lack of labeled anomaly instances available for training.

Another major challenge in log-based anomaly detection is stream processing. Log data is generated as a continuous stream of data, which requires deep learning systems to be designed for single-pass data processing to enable real-time monitoring rather than forensic analysis. The sheer volume of log data generated by many systems requires efficient algorithms to ensure real-time processing, especially on machines with limited computational resources.

Finally, model explainability can also be a challenge with approaches based on neural networks, making it difficult to understand the reasons behind both correct and incorrect classifications. Addressing these challenges requires a combination of data preprocessing techniques, model design, algorithm development, and the use of new techniques for model explainability and interpretability.

RESEARCH QUESTIONS

In recent years, the use of pre-trained language models such as BERT and GPT has become increasingly popular in various applications, including log-based anomaly detection. Understanding the performance of these models in this context is crucial for ensuring that they are effectively and accurately detecting unusual patterns. In addition, it is important to understand the limitations of these models, as this will allow us to identify areas for improvement and develop more effective methods for log-based anomaly detection.

One of the challenges of using pre-trained language models for anomaly detection in server logs is their explainability. Explainability refers to the ability of a model to provide clear and interpretable insights into the reasoning behind its predictions. Improving the explainability of these models is crucial for ensuring that the results of log-based anomaly detection are easily understandable and actionable for end-users.

Another challenge is the limitations of pre-trained language models for anomaly detection in server logs. These models may struggle to handle complex and diverse server logs, which can result in inaccurate or incomplete predictions. Mitigating these limitations is crucial for improving the overall performance of pre-trained language models for anomaly detection in server logs.

While these models can identify patterns and relationships within the data, they may struggle to understand the full context of each log entry. This can result in inaccurate or incomplete predictions, particularly when dealing with complex or diverse server logs.

Contextualization is critical for anomaly detection because it helps the model understand the broader context in which each log entry occurs. For example, a pre-trained language model may struggle to identify an unusual pattern of requests from a particular IP address if it does not have access to information about the type of server, the time of day, or other relevant factors. Similarly, the model may struggle to detect anomalies in logs that contain multiple types of entries or data from different sources.

To address these limitations, researchers are exploring new techniques for pre-processing

server logs to improve the quality of data fed into pre-trained language models. One approach is to use log parsing tools that can identify different types of log entries and group them together based on their context. This can help pre-trained language models to better understand the structure of server logs and identify anomalies more accurately.

Another approach is to use domain-specific language models that are trained on data from specific types of servers or applications. These models can be more effective at contextualizing server logs and identifying anomalies that are specific to the domain in question.

Overall, improving the ability of pre-trained language models to contextualize server logs is an ongoing challenge for the field of anomaly detection. By developing new pre-processing techniques and domain-specific language models, researchers hope to improve the accuracy and completeness of predictions and make anomaly detection more effective in a wide range of applications.

In conclusion, understanding the performance, limitations, and explainability of pre-trained language models for anomaly detection in server logs is crucial for ensuring that they are effectively detecting unusual patterns and providing actionable insights. The results of this research will contribute to the development of more effective methods for log-based anomaly detection and help ensure that these models are used to their full potential in this context.

EXPERIMENTAL SETUP AND STUDY

Tools

Openai Api

The OpenAI API is a versatile tool that can be applied to a wide range of tasks related to natural language, code, and images. It is designed to be easy to use and integrate into existing workflows by developers and researchers working with NLP applications. It offers several pre-trained models, including GPT-3, which is capable of generating natural language text in response to a given prompt. The API also offers tools for fine-tuning and customizing these models to specific use cases and training new models from scratch.

Text-davinci-003

Text-Davinci-003 is a language model developed by OpenAI and is one of the latest additions to the GPT -3 family of models. It is an incredibly powerful language model with a massive 175 billion parameters, making it one of the largest and most advanced language models in existence. Text-Davinci-003 was specifically designed to excel at instruction-following tasks, and its capabilities extend to a wide range of natural language processing applications, including machine translation, language understanding, and text generation. Its ability to generate accurate and concise responses, even in zero-shot scenarios, makes it an invaluable tool for businesses and organizations looking to leverage the power of AI in their operations.

Datasets

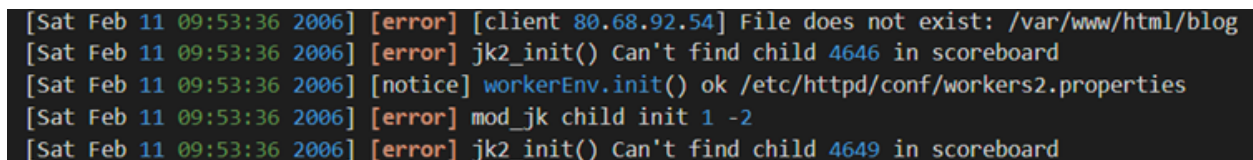
Apache

The Apache web server is one of the most widely used web servers in the world. It is open-source software that can run on multiple platforms and is used to serve web pages over the internet. Apache web server logs are records of events that occur on the server, including requests

from clients, errors, and security-related events.

Apache web server logs are essential for monitoring and troubleshooting the server's performance and security. These logs provide detailed information about the requests received by the server, including the client's IP address, the requested URL, the response code, and the number of bytes sent. By analyzing Apache web server logs, system administrators can identify performance bottlenecks, troubleshoot errors, and detect potential security breaches. For example, they can identify URLs generating high traffic or requests that take a long time to complete.

Apache web server logs can also be used to generate statistics about the server's usage patterns, including the number of requests, the amount of data transferred, and the most frequently requested URLs. This information can be used to optimize the server's performance and capacity planning.



```
[Sat Feb 11 09:53:36 2006] [error] [client 80.68.92.54] File does not exist: /var/www/html/blog
[Sat Feb 11 09:53:36 2006] [error] jk2_init() Can't find child 4646 in scoreboard
[Sat Feb 11 09:53:36 2006] [notice] workerEnv.init() ok /etc/httpd/conf/workers2.properties
[Sat Feb 11 09:53:36 2006] [error] mod_jk child init 1 -2
[Sat Feb 11 09:53:36 2006] [error] jk2_init() Can't find child 4649 in scoreboard
```

Figure 7. Apache Web Server Logs

Openssh

OpenSSH is a free and open-source implementation of the SSH (Secure Shell) protocol, which was originally developed by Tatu Ylonen. The OpenBSD team and the user community have since further developed OpenSSH, based on the free version.

While Tatu Ylonen founded SSH Communications Security to offer commercial support for enterprises, the original version evolved into Tectia SSH. The commercial version of SSH also provides support for Windows and IBM mainframe (z/OS) platforms, and includes full support for X.509 certificates and smartcard authentication, such as the CAC and PIV cards used by the US government. Analyzing OpenSSH logs can help system administrators identify suspicious or unauthorized activities, such as brute force attacks, password guessing, or attempts to exploit vulnerabilities. By reviewing the logs, administrators can take appropriate measures to prevent

unauthorized access and protect their systems from potential security breaches.

Moreover, OpenSSH logs can also assist in compliance with regulatory standards by documenting user activity and providing evidence for audits. By maintaining accurate and detailed logs, organizations can demonstrate their adherence to security policies and compliance requirements.

In summary, OpenSSH logs are a critical resource for monitoring, securing, and auditing SSH activity on a system. As a system administrator, it's essential to regularly review these logs and tailor them to your specific needs and requirements.

```
Dec 10 10:57:16 LabSZ sshd[25047]: Received disconnect from 183.62.140.253: 11: Bye Bye [preauth]
Dec 10 10:57:17 LabSZ sshd[25049]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=183.62.140.253
Dec 10 10:57:19 LabSZ sshd[25049]: Failed password for root from 183.62.140.253 port 37388 ssh2
Dec 10 10:57:19 LabSZ sshd[25049]: Received disconnect from 183.62.140.253: 11: Bye Bye [preauth]
Dec 10 10:57:19 LabSZ sshd[25052]: pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0 tty=ssh ruser= rhost=183.62.140.253
Dec 10 10:57:22 LabSZ sshd[25052]: Failed password for root from 183.62.140.253 port 37894 ssh2
```

Figure 8. OpenSSH Server Logs

Automated Log Parsing

According to Zhu, He, Liu, et al. (2019), “The goal of log parsing is to convert each log message into a specific event template (e.g., “Received block i^*l of size i^*l from $/i^*l$ ”) associated with key parameters (e.g., [“blk_562725280853087685”, “67108864”, “10.251.91.84”]). *Here, $\backslash < * >$ denote the position of each parameter.*”).

Automated log parsing is the process of extracting structured information from textual log messages. This allows for efficient search, filtering, grouping, counting, and mining of logs. To evaluate the effectiveness of automated log parsing, a benchmarking process is often used. To ensure that the benchmarking results are reproducible, a sample of log messages is randomly selected from each dataset, and event templates are manually labeled as ground truth.

The parsing accuracy (PA) metric is then used to quantify the effectiveness of the automated log parser. PA is defined as the ratio of correctly parsed log messages over the total number of log messages. After parsing, each log message is assigned an event template, which corresponds to a group of messages having the same template. If the event template corresponds to the same group of log messages as the ground truth, the log message is considered to be parsed correctly. Partially

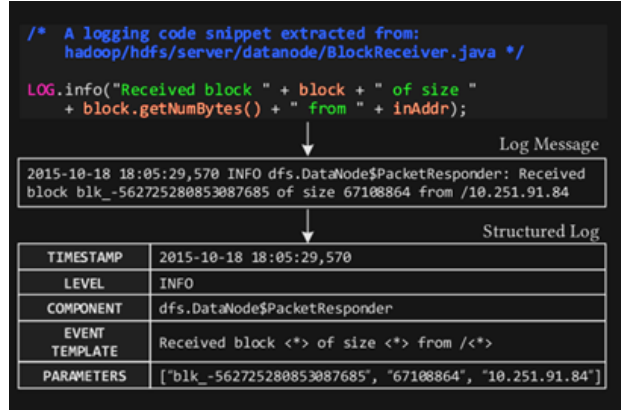


Figure 9. Log Parsing

matched events are considered incorrect in the PA metric.

The number of event templates generated by the rule-based log parser is compared with the number of event templates in the 2k sample to determine the effectiveness of the parser. This approach allows researchers and practitioners to compare different automated log parsers’ effectiveness and select the one that performs best for their specific application. Overall, automated log parsing and benchmarking are essential tools for analyzing large volumes of logs efficiently. They enable researchers and practitioners to gain insights into system behavior and detect anomalies quickly.

To understand the effectiveness of OpenAi’s API for log parsing, logs sampled by Zhu, He, Liu, et al. (2019) were used, and the authors also manually assigned event templates for the sampled logs. The logs parsed by OpenAI API were matched with the templates using a regex matching algorithm inspired by the Zhu, He, Liu, et al. (2019). Then inner join between matched logs and the ground truth file was applied to identify the number of logs that had been parsed correctly.

OpenAi’s “text-DaVinci-3” model was used to parse Apache logs, and the following prompt was given. Prompt: ” Parse the following Apache log entry and extract any relevant fields:- time - level - content. For example this apache log -[Thu Jun 09 06:07:04 2005] [notice] LDAP: Built with OpenLDAP LDAP SDK can be parsed as Time:[Thu Jun 09 06:07:04 2005], Level:[notice], Content: LDAP: Built with OpenLDAP LDAP SDK ”

```

# Apache2klog
1 [Sun Dec 04 04:47:44 2005] [notice] workerEnv.init() ok /etc/httpd/conf/workers2.properties
2 [Sun Dec 04 04:47:44 2005] [error] mod_jk child workerEnv in error state 6
3 [Sun Dec 04 04:51:08 2005] [notice] jk2_init() Found child 6725 in scoreboard slot 10
4 [Sun Dec 04 04:51:09 2005] [notice] jk2_init() Found child 6726 in scoreboard slot 8
5 [Sun Dec 04 04:51:09 2005] [notice] jk2_init() Found child 6728 in scoreboard slot 6
6 [Sun Dec 04 04:51:14 2005] [notice] workerEnv.init() ok /etc/httpd/conf/workers2.properties
7 [Sun Dec 04 04:51:14 2005] [notice] workerEnv.init() ok /etc/httpd/conf/workers2.properties
8 [Sun Dec 04 04:51:14 2005] [notice] workerEnv.init() ok /etc/httpd/conf/workers2.properties
9 [Sun Dec 04 04:51:18 2005] [error] mod_jk child workerEnv in error state 6
10 [Sun Dec 04 04:51:18 2005] [error] mod_jk child workerEnv in error state 6
11 [Sun Dec 04 04:51:18 2005] [error] mod_jk child workerEnv in error state 6

```

Figure 10. Original Log File

```

# apache_parsed_logslog
1 { "Time": [Sun Dec 04 04:47:44 2005], "level": [notice], "Content": workerEnv.init() ok /etc/httpd/conf/workers2.properties }
2 { "Time": [Sun Dec 04 04:47:44 2005], "level": [error], "Content": mod_jk child workerEnv in error state 6 }
3 { "Time": [Sun Dec 04 04:51:08 2005], "level": [notice], "Content": jk2_init() Found child 6725 in scoreboard slot 10 }
4 { "Time": [Sun Dec 04 04:51:09 2005], "level": [notice], "Content": jk2_init() Found child 6726 in scoreboard slot 8 }
5 { "Time": [Sun Dec 04 04:51:09 2005], "level": [notice], "Content": jk2_init() Found child 6728 in scoreboard slot 6 }
6 { "Time": [Sun Dec 04 04:51:14 2005], "level": [notice], "Content": workerEnv.init() ok /etc/httpd/conf/workers2.properties }
7 { "Time": [Sun Dec 04 04:51:14 2005], "level": [notice], "Content": workerEnv.init() ok /etc/httpd/conf/workers2.properties }
8 { "Time": [Sun Dec 04 04:51:14 2005], "level": [notice], "Content": workerEnv.init() ok /etc/httpd/conf/workers2.properties }
9 { "Time": [Sun Dec 04 04:51:18 2005], "level": [error], "Content": mod_jk child workerEnv in error state 6 }
10 { "Time": [Sun Dec 04 04:51:18 2005], "level": [error], "Content": mod_jk child workerEnv in error state 6 }
11 { "Time": [Sun Dec 04 04:51:18 2005], "level": [error], "Content": mod_jk child workerEnv in error state 6 }

```

Figure 11. Parsed Logs

	LineId	Time	...	EventId	EventTemplate
0	1	Sun Dec 04 04:47:44 2005	...	E2	workerEnv.init() ok <*>
1	2	Sun Dec 04 04:47:44 2005	...	E3	mod_jk child workerEnv in error state <*>
2	3	Sun Dec 04 04:51:08 2005	...	E1	jk2_init() Found child <*> in scoreboard slot <*>
3	4	Sun Dec 04 04:51:09 2005	...	E1	jk2_init() Found child <*> in scoreboard slot <*>
4	5	Sun Dec 04 04:51:09 2005	...	E1	jk2_init() Found child <*> in scoreboard slot <*>

Figure 12. Loading Ground Truth File into the data frame

0	1	Sun Dec 04 04:47:44 2005	...	E2	[/, e, t, c, /, h, t, t, p, d, /, c, o, n, f, ...
1	2	Sun Dec 04 04:47:44 2005	...	E3	[6]
2	3	Sun Dec 04 04:51:08 2005	...	E1	[6725, 10]
3	4	Sun Dec 04 04:51:09 2005	...	E1	[6726, 8]
4	5	Sun Dec 04 04:51:09 2005	...	E1	[6728, 6]

Figure 13. Log Parsing

	LineId	Time_x	...	Content_y	EventTemplate
0	1	Sun Dec 04 04:47:44 2005	...	workerEnv.init() ok /etc/httpd/conf/workers2.p...	workerEnv.init() ok <*>
1	2	Sun Dec 04 04:47:44 2005	...	mod_jk child workerEnv in error state 6	mod_jk child workerEnv in error state <*>
2	3	Sun Dec 04 04:51:08 2005	...	jk2_init() Found child 6725 in scoreboard slot 10	jk2_init() Found child <*> in scoreboard slot <*>
3	4	Sun Dec 04 04:51:09 2005	...	jk2_init() Found child 6726 in scoreboard slot 8	jk2_init() Found child <*> in scoreboard slot <*>
4	5	Sun Dec 04 04:51:09 2005	...	jk2_init() Found child 6728 in scoreboard slot 6	jk2_init() Found child <*> in scoreboard slot <*>
...
1988	1996	Mon Dec 05 19:14:11 2005	...	mod_jk child workerEnv in error state 6	mod_jk child workerEnv in error state <*>
1989	1997	Mon Dec 05 19:15:55 2005	...	jk2_init() Found child 6791 in scoreboard slot 8	jk2_init() Found child <*> in scoreboard slot <*>
1990	1998	Mon Dec 05 19:15:55 2005	...	jk2_init() Found child 6790 in scoreboard slot 7	jk2_init() Found child <*> in scoreboard slot <*>
1991	1999	Mon Dec 05 19:15:57 2005	...	workerEnv.init() ok /etc/httpd/conf/workers2.p...	workerEnv.init() ok <*>
1992	2000	Mon Dec 05 19:15:57 2005	...	mod_jk child workerEnv in error state 6	mod_jk child workerEnv in error state <*>

[1993 rows x 6 columns]
99.65% logs are parsed correctly.

Figure 14. Log Parsing

Other hyperparameters were set as follows: `max_tokens` = 300, `n` (no.of completions to generate) - 1, `temperature` = 0. According to OpenAI, “Temperature is a parameter of OpenAI ChatGPT, GPT-3 and GPT-4 models that govern the randomness and thus the creation of the responses.”, `stop` = 'None' (no strings to avoid). While using the model for log parsing, the temperature was the most important variable because the change in the temperature hyperparameter’s value resulted in a drastic change in parsed logs. To keep the format of parsed logs predictable, the temperature had to be set to 0. It still deviated from the format in rare occurrences, and these cases were handled using exception handling and conditions.

The parsing accuracy of Apache logs was 99.65% for the sampled logs, which is near to the best case of 100%. As Apache and HDFS logs have relatively fewer and simpler log templates, they are easily parsed with many log parsers with high parsing accuracy. The text-DaVinci-3 model has a parsing accuracy of 99.25% on OpenSSH logs which have around 5x more log templates which are relatively more complex. It is important to note that the parsing accuracy is best; the model performed better than 13 other parsers using different type of techniques, including clustering, heuristics, frequent pattern matching, iterative portioning etc.

Table 1. Comparing log parsers

Parsing Technique	Apache	OpenSSH
SLCT	0.731	0.521
AEL	1	0.538
IPLOM	1	0.802
LKE	1	0.426
LFA	1	0.501
LogSig	0.582	0.373
SHISO	1	0.619
LogCluster	0.709	0.426
LenMa	1	0.925
LogMine	1	0.431
Spell	1	0.554
Drain	1	0.788
MoLFI	1	0.5
Text-Davinci-3	0.996	0.992

One advantage of using OpenAI API’s models for log parsing is that they offer greater

flexibility in adapting to changes in log formats. This can be particularly useful in situations where log formats are changing frequently or where different types of log messages require different parsing rules or techniques. With OpenAI API's models, it may be possible to train the model on new log formats without needing to manually update parsing rules or make other changes to the parsing system. It can be observed by changing the prompt to "Parse the following Apache log entry and extract any relevant fields:- - Date - Time - Log Level - Module - Client IP Address (if present) - HTTP Request Method (if present)- HTTP Request Path (if present)- Message".

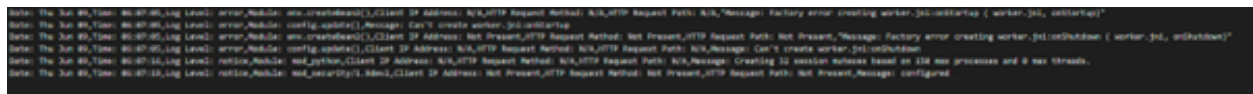


Figure 15. Comparison with 13 other parsers

While traditional machine learning models are trained on large datasets consisting of thousands or millions of examples, which they use to learn patterns and features that help them classify new instances. The text-davinci-3 can effectively learn from a single example in the prompt. As we observe in Prompt: "Parse the following Apache log entry and extract any relevant fields:- time - level - content. For example, this apache log -[Thu Jun 09 06:07:04 2005] [notice] LDAP: Built with OpenLDAP LDAP SDK can be parsed as Time:[Thu Jun 09 06:07:04 2005], Level:[notice], Content: LDAP: Built with OpenLDAP LDAP SDK" the model can be easily trained to do a task based on limited data. The ability of an algorithm to recognize a new concept/object based on just one example is known as one-shot learning. It is particularly useful when obtaining large amounts of training data is difficult or expensive or where the objects or concepts being classified are rare or unique.

Challenges

GPT (Generative Pre-trained Transformer) is a powerful language model that can generate text by predicting the most likely next word or sequence of words given a context. This context can be a prompt or previous text input.

When it comes to log analysis, GPT-3 can be used to extract relevant fields from log mes-

sages by predicting which fields are most likely to appear based on the context of the log message. However, this approach can lead to problems, particularly when GPT-3 is inferring missing information based on previous log messages.

For example, we have a log message indicating that an SSH connection was established with a user "root." In the next log message, GPT-3 might infer that "root" is still the active user even if the second message does not explicitly state the user. This is because GPT-3 is trained to recognize patterns in language and predict likely continuations based on that context.

```
{ "Date":Dec,"Day":10,"Time": 07:07:38,"Component": LabSZ,"Pid": 24206,"Content": pam_unix(sshd:auth): check pass; user unknown}
user=root

{ "Date":Dec,"Day":10,"Time": 07:07:38,"Component": LabSZ,"Pid": 24206,"Content": pam_unix(sshd:auth): authentication failure; logname=
{ "Date":Dec,"Day":10,"Time": 07:07:45,"Component": LabSZ,"Pid": 24206,"Content": Failed password for invalid user test9 from 52.80.34.1
{ "Date":Dec,"Day":10,"Time": 07:07:45,"Component": LabSZ,"Pid": 24206,"Content": Received disconnect from 52.80.34.196: 11: Bye Bye [pr
{ "Date":Dec,"Day":10,"Time": 07:08:28,"Component": LabSZ,"Pid": 24208,"Content": reverse mapping checking getaddrinfo for ns.marryaldkf
{ "Date":Dec,"Day":10,"Time": 07:08:28,"Component": LabSZ,"Pid": 24208,"Content": Invalid user webmaster from 173.234.31.186}
{ "Date":Dec,"Day":10,"Time": 07:08:28,"Component": LabSZ,"Pid": 24208,"Content": input_userauth_request: invalid user webmaster [preaut
{ "Date":Dec,"Day":10,"Time": 07:08:28,"Component": LabSZ,"Pid": 24208,"Content": pam_unix(sshd:auth): check pass; user unknown}
user=root

{ "Date":Dec,"Day":10,"Time": 07:08:28,"Component": LabSZ,"Pid": 24208,"Content": pam_unix(sshd:auth): authentication failure; logname=
```

Figure 16. Unnecessary Inference 'user=root'

However, this kind of inference can lead to errors or incorrect assumptions, especially if the previous log message is unrelated to the current one or if the context has changed somehow. For example, if the first log message was from an hour ago, the SSH connection has since been closed, and a different user makes a new connection, GPT-3 might still assume that the "root" is the active user based on the previous log message. This can lead to incorrect analysis and potentially missed security threats.

To mitigate this problem, it is important to carefully consider the context of each log message and evaluate the accuracy of the extracted fields. It is also a good practice to periodically review the system and update the log analysis tools and models to ensure they are accurately reflecting the current state of the system. Additionally, using a combination of different log analysis techniques, such as regular expressions and custom machine learning models, can help improve the accuracy and reliability of log analysis.

Deviations in the log message format can pose a challenge for any log analysis tool. If the log messages deviate significantly from the expected format, it can cause errors or inaccuracies in

```
{ "Date":Dec,"Day":10,"Time": 07:07:38,"Component": LabSZ,"Pid": 24206,"Content": pam_unix(sshd:auth): check pass; user unknown}
user=root

{ "Date":Dec,"Day":10,"Time": 07:07:38,"Component": LabSZ,"Pid": 24206,"Content": pam_unix(sshd:auth): authentication failure; logname=
{ "Date":Dec,"Day":10,"Time": 07:07:45,"Component": LabSZ,"Pid": 24206,"Content": Failed password for invalid user test9 from 52.80.34.1
{ "Date":Dec,"Day":10,"Time": 07:07:45,"Component": LabSZ,"Pid": 24206,"Content": Received disconnect from 52.80.34.196: 11: Bye Bye [pr
{ "Date":Dec,"Day":10,"Time": 07:08:28,"Component": LabSZ,"Pid": 24208,"Content": reverse mapping checking getaddrinfo for ns.marryaldf
{ "Date":Dec,"Day":10,"Time": 07:08:28,"Component": LabSZ,"Pid": 24208,"Content": Invalid user webmaster from 173.234.31.186}
{ "Date":Dec,"Day":10,"Time": 07:08:28,"Component": LabSZ,"Pid": 24208,"Content": input_userauth_request: invalid user webmaster [preaut
{ "Date":Dec,"Day":10,"Time": 07:08:28,"Component": LabSZ,"Pid": 24208,"Content": pam_unix(sshd:auth): check pass; user unknown}
user=root

{ "Date":Dec,"Day":10,"Time": 07:08:28,"Component": LabSZ,"Pid": 24208,"Content": pam_unix(sshd:auth): authentication failure; logname=
```

Figure 17. Deviation in log format

the extracted fields. To handle deviations in format, it's important to have a flexible and robust parsing system that can handle a variety of different formats and edge cases. This might involve using a combination of regular expressions, custom parsing rules, and machine learning models to extract the relevant information from each log message accurately.

Filtering and pre-processing can be effective techniques for handling minor errors or inconsistencies in log messages. These techniques can help to clean and normalize the log data, making it easier to extract accurate information. As observed in the following Apache logs, in a few instances, while processing a log with a relatively higher token length, the parser deviates from the standard format. To handle these cases, exception handling can be used, or logs should be filtered based on conditions.

```
# Check if parsed log matches expected format
if '"Time\":[\'' in parsed_log and '"Level\":[\'' in parsed_log and '"Content\":[\'' in parsed_log:
    # Extract time, level and content from parsed log
    time_str = str(parsed_log.split('"Time\":[\''')[1].split(',')[0])
    level_str = str(parsed_log.split('"Level\":[\''')[1].split(',')[0])
    content_str = str(parsed_log.split('"Content\":[\''')[1].strip().rstrip(','))

    # Format log in desired format
    formatted_log = f"[{time_str}] [{level_str}] {content_str}"
else:
    # Return original log message without formatting
    formatted_log = "[null] [null] no content"

with open('apache_parsed_logs.log', "a") as parsed_logs_file:
    parsed_logs_file.write(parsed_log + "\n")

return formatted_log
except Exception as e:
    # Log any errors that occur
    print(str(e))
    return "[null] [null] no content"
```

Figure 18. Deviation in log format

Filtering involves removing or excluding log messages that do not meet certain criteria or conditions. For example, if the log messages contain a specific keyword or pattern that indicates a

particular type of event, filtering can only extract those messages that meet those criteria. This can help to reduce noise and improve the accuracy of the extracted fields.

Pre-processing, on the other hand, involves transforming or normalizing the log data before it is analysed. This might involve tasks such as removing unnecessary whitespace, converting the data to a specific format or encoding, or replacing certain characters or strings with more standardized versions. These pre-processing steps can help ensure the log data is consistent and standardized, making it easier to extract accurate information using tools like OpenAI API.

It's worth noting, however, that filtering and pre-processing should be used judiciously and with caution. Overly aggressive filtering or pre-processing can result in lost or distorted data, which can in turn lead to inaccurate analysis and missed security threats. As with any log analysis technique, it's important to carefully evaluate the trade-offs and ensure that the filtering and pre-processing steps are appropriate for the specific use case and the system being analysed.

In the case of text-davinci-3, it may be necessary to train the model on a diverse set of log messages that include a range of different formats and variations. This can help the model learn to recognize and handle deviations in format more effectively. Additionally, it's essential to monitor the performance of the log analysis tool and adjust it as needed to ensure accurate and reliable results. This might involve periodically reviewing the extracted fields to identify any errors or inaccuracies, and adjusting the parsing rules or model as needed.

REFERENCES

- Chen, Z., Liu, J., Gu, W., Su, Y., & Lyu, M. R. (2021). Experience report: Deep learning-based system log analysis for anomaly detection. <https://doi.org/10.48550/ARXIV.2107.05908>
- Dobilas, S. (2022). Lstm recurrent neural networks — how to teach a network to remember the past. <https://towardsdatascience.com/lstm-recurrent-neural-networks-how-to-teach-a-network-to-remember-the-past-55e54c2ff22e>
- Du, M., Li, F., Zheng, G., & Srikumar, V. (2017). Deeplog: Anomaly detection and diagnosis from system logs through deep learning. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*.
- Guo, H., Yuan, S., & Wu, X. (2021). Logbert: Log anomaly detection via BERT. *CoRR*, abs/2103.04475. <https://arxiv.org/abs/2103.04475>
- Halthor, A. (2020). Bert neural network - explained!
- He, P., Zhu, J., He, S., Li, J., & Lyu, M. R. (2016). An evaluation study on log parsing and its use in log mining. *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 654–661. <https://doi.org/10.1109/DSN.2016.66>
- He, S., Zhu, J., He, P., & Lyu, M. R. (2020). Loghub: A large collection of system log datasets towards automated log analytics. <https://doi.org/10.48550/ARXIV.2008.06448>
- Huo, Y., Su, Y., Lee, C., & Lyu, M. R. (2021). Semparser: A semantic parser for log analysis. <https://doi.org/10.48550/ARXIV.2112.12636>
- Landauer, M., Onder, S., Skopik, F., & Wurzenberger, M. (2022). Deep learning for anomaly detection in log data: A survey.
- Le, V.-H., & Zhang, H. (2022). Log-based anomaly detection with deep learning: How far are we? *Proceedings of the 44th International Conference on Software Engineering*. <https://doi.org/10.1145/3510003.3510155>
- Pang, G., Shen, C., Cao, L., & Hengel, A. V. D. (2021). Deep learning for anomaly detection: A review. *ACM Comput. Surv.*, 54(2). <https://doi.org/10.1145/3439950>

- Thudumu, S., Branch, P., Jin, J., & Singh, J. (2020). A comprehensive survey of anomaly detection techniques for high dimensional big data. *Journal of Big Data*, 7. <https://doi.org/10.1186/s40537-020-00320-x>
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. <https://arxiv.org/pdf/1706.03762.pdf>
- Zhu, J., He, S., Liu, J., He, P., Xie, Q., Zheng, Z., & Lyu, M. R. (2019). Tools and benchmarks for automated log parsing.

APPENDIX A: CODE AND IMAGES

```

Loading log messages to dataframe...
Loading 1999 messages done, loading rate: 100.0%
Processing log file: ssh formatted logs.log
Loading log messages to dataframe...
Loading 1999 messages done, loading rate: 100.0%
Matching event templates...
Matching 1999 lines.

Matching done, matching rate: 100.0% [Time taken: 0:00:01.057993]

```

LineId	Date	Day	Time	Component	Pid	Content	EventId	ParameterList
0	1	Dec	10 06:55:46	LabSZ sshd	24200	reverse mapping checking getaddrinfo for ns.ma...	E27	[ns.marryaldkfaczcz.com, 173.234.31.186]
1	2	Dec	10 06:55:46	LabSZ sshd	24200	Invalid user webmaster from 173.234.31.186	E13	[webmaster, 1]
2	3	Dec	10 06:55:46	LabSZ sshd	24200	input_userauth_request: invalid user webmaster...	E12	[w, e, b, m, a, s, t, e, r]
3	4	Dec	10 06:55:46	LabSZ sshd	24200	pam_unix(sshd:auth): check pass; user unknown	E21	[p, a, m, _ , u, n, i, x, (, s, s, h, d, :, a, ...]
4	5	Dec	10 06:55:46	LabSZ sshd	24200	pam_unix(sshd:auth): authentication failure; l...	E19	[0, 0, 1]
...
1994	1996	Dec	10 11:04:42	LabSZ sshd	25539	pam_unix(sshd:auth): authentication failure; l...	E19	[0, 0, 1]
1995	1997	Dec	10 11:04:43	LabSZ sshd	25541	Failed password for root from 183.62.140.253 p...	E9	[root, 183.62.140.253, 36300]
1996	1998	Dec	10 11:04:43	LabSZ sshd	25541	Received disconnect from 183.62.140.253: 11: B...	E24	[183.62.140.253, 11]
1997	1999	Dec	10 11:04:43	LabSZ sshd	25544	pam_unix(sshd:auth): authentication failure; l...	E20	[1]
1998	2000	Dec	10 11:04:45	LabSZ sshd	25539	Failed password for invalid user user from 103...	E10	[user, 103.99.0.122, 52683, ssh2]

```

[1999 rows x 9 columns]

```

LineId	Date	Day	Time	...	Pid	Content	EventTemplate	merge
0	1	Dec	10 06:55:46	...	24200	reverse mapping checking getaddrinfo for <*> [...	reverse mapping checking getaddrinfo for <*> [...	both
1	2	Dec	10 06:55:46	...	24200	Invalid user webmaster from 173.234.31.186	Invalid user <*> from <*>	both
2	3	Dec	10 06:55:46	...	24200	input_userauth_request: invalid user webmaster...	input_userauth_request: invalid user <*> [<*>]	both
3	4	Dec	10 06:55:46	...	24200	pam_unix(sshd:auth): check pass; user unknown	pam_unix(sshd:auth): check pass; user <*>	both
4	5	Dec	10 06:55:46	...	24200	pam_unix(sshd:auth): authentication failure; l...	pam_unix(sshd:auth): authentication failure; l...	both
...
1980	1996	Dec	10 11:04:42	...	25539	pam_unix(sshd:auth): authentication failure; l...	pam_unix(sshd:auth): authentication failure; l...	both
1981	1997	Dec	10 11:04:43	...	25541	Failed password for root from 183.62.140.253 p...	Failed password for <*> from <*> port <*> <*>	both
1982	1998	Dec	10 11:04:43	...	25541	Received disconnect from 183.62.140.253: 11: B...	Received disconnect from <*>: <*>: Bye Bye [<*>]	both
1983	1999	Dec	10 11:04:43	...	25544	pam_unix(sshd:auth): authentication failure; l...	pam_unix(sshd:auth): authentication failure; l...	both
1984	2000	Dec	10 11:04:45	...	25539	Failed password for invalid user user from 103...	Failed password for invalid user <*> from <*> ...	both

```

[1985 rows x 18 columns]
1985
99.25% logs are parsed correctly.

```

Figure 19. SSH Log Parsing

Code

Use the following link to access the code.

<https://github.com/startsmiit/Thesis>

The repository contains 2 log parsers, 2 log loaders, 6 dataset files (2 dataset files, ground 2 truth files, and 2 template files for Apache and OpenSSH datasets), and output files. To test, run the parser by adding api key and editing the directory location in the files. To check the results, use the log loaders. Log loaders do preprocessing of the parsed logs and match them with templates using a regex matching algorithm. In end, parsing accuracy is calculated by comparing matched logs with ground truth (structured log files).