Semantic Log Partitioning: Towards Automated Root Cause Analysis

Bahareh Afshinpour, Massih-Reza Amini, Roland Groz

Semantic Log Partitioning: Towards Automated Root Cause Analysis

Bahareh Afshinpour, Massih-Reza Amini, and Roland Groz
Université Grenoble Alpes, INP, CNRS
Computer Science Laboratory (LIG)
CS 40700 38058 Grenoble Cedex 9, France
{Firstname.Lastname}@univ-grenoble-alpes.fr

Abstract—In recent years, the significance of test logs in ensuring system reliability and diagnosing runtime events has grown significantly, particularly with software expanding into various domains, necessitating rigorous verification and validation processes. However, the complexity and cost of testing have prompted a shift towards automation. This paper addresses the challenges of automated software testing through root-cause event detection. The proposed approach initially involves parsing and partitioning logs, followed by representing test events as dense vectors in a continuous space, enabling the capture of semantic similarities and relationships among events based on their sequence positions. Subsequently, test events are clustered in this embedded space, and each log partition is represented as a vector, with its characteristics reflecting the number of events in the log partition present in the clusters. Through two distinct case studies, we demonstrate that the final clustering of log partitions in this new space efficiently identifies root cause events. We evaluate our approach on two applications and anticipate its contribution as a cornerstone for future research and deployment of automated log mining.

Keywords–Automated software testing; Log analysis; Machine learning; Log mining task; Root-cause events detection

1. Introduction

Software testing is one of the most important phases of the software development lifecycle. It is used to detect software flaws and ensure that software is delivered in a high-quality condition. Any changes to a software component may affect one or more other components, requiring the re-execution of previously generated test cases in addition to the newly generated ones [1].

Regression testing is a software testing technique that verifies that an application continues to perform as expected after any code modifications, updates, or improvements. It should be performed after each iteration of software development, which can be expensive in terms of time and resources.

Testing information systems has become a serious bottleneck for many large corporations and small and medium-sized enterprises. Aside from the ever-increasing complexity of such systems, their unavoidable quality assurance requirements have resulted in drastically increased verification and validation expenses. It is thus critical to provide more intelligent and automatic automated test processes in order to regulate the complexity and growth of software verification activities and help to break the testing bottleneck. This will allow test engineers to concentrate on developing higher-value tests for specific scenarios.

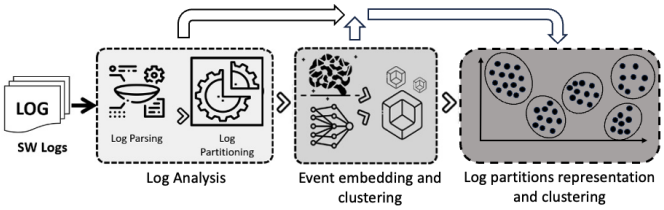

Figure 1: Illustration of the three phases of the proposed approach.

The traditional method of log mining, which focused in error keywords, has evolved over the last two decades into a rule-based comparison approach [2]. However, this method is prone to errors, and may not capture the full range of anomalies or issues present in complex systems [3]. Furthermore, as systems grow larger and more complex, the scalability of traditional methods becomes a significant concern [4]. Rule-based approaches may struggle to handle the increasing volume and diversity of log data efficiently [5]. These approaches are also often based on human-defined rules, which can be subjective and may not generalize well across different systems or environments. This subjectivity can lead to inaccuracies and inconsistencies in the identification of root causes. With advancements in machine learning, log mining has explored new dimensions in log analysis [6].

This paper introduces `TRAIL`, a novel and efficienT Root cAuse Identification through Log partition clustering approach from raw log files. Our methodology relies on unsupervised learning applied to raw log files, and it comprises three main phases: log analysis, representation of test events and log-partitions, and root-cause detection. Each phase requires customization to adapt to different software systems and logging scenarios. A key aspect of our methodology involves clustering at two stages. Initially, we partition test events in an embedded space that captures semantic similarities and relationships among events based on their contextual cues in the logs. Subsequently, each log partition is represented as a vector, where its attributes are proportional to the number of events within the log partition present in the clusters of test events. We validate our approach through an examination of two distinct case studies. Figure 1 provides an overview of these three phases.

We first apply our approach to a large record of incoming events from an internet access device over six months, where

several system failure occurrences were attributed to specific tests during the testing phase. Furthermore, we extend our method to the Scanner case study - a device used in supermarkets to read product barcodes and create shopping lists for self-service purchases. We demonstrate how the clustering of log partitions effectively identifies root cause events on these two case studies. We also utilized our method to reduce extensive user trace logs into smaller ones with similar bug-triggering effects, effectively selecting a subset of text events from the log.

Ultimately, for research purposes and reproducibility, we have made our codes publicly available as two open-source tools[1]. Therefore, the main contribution of this paper is twofold:

- It introduces a new unsupervised methodology for detecting root cause events from raw log files. The methodology is adaptable to different software systems and logging scenarios and does not require human annotation by employing clustering techniques. By partitioning system events into clusters, it provides deeper insights into log files, including causal relationships. Furthermore, it realizes that the proposed methodology is adopted to reduce enormous user trace logs to smaller ones.
- The approach is tested across two different case studies featuring unconventional logging styles aimed at monitoring purposes, demonstrating its efficacy in detecting root cause events. Furthermore, the codes are made publicly available as two open-source tools, thereby improving accessibility and facilitating reproducibility.

In the rest of the paper, we explore related approaches to this work in Section 2. Section 3 details our approach, followed by the experimental results in Section 5. we outline in Section 4 the two case studies considered in this research. And, finally present our conclusion and perspectives in Section 6.

2. Related work

The classical viewpoint on software testing assumes that for each given input entry, the software returns an output (or a log event) record which are distinct from the other input-output (or input-log-event) pairs. Accordingly, assigning “Pass” or “Fail” labels to the output logs is mostly feasible based on the input and the expected software functionality. These separated “input-output” or “input-log” pairs form a basis to test a software artifact or perform some post-processing steps on test-suites, like “regression testing”, “test-suite reduction (TSR)” [7] or bug prediction [8] . From this perspective, the effect of a single or a set of inputs is mapped to a limited set of outputs or log events. A shopping software is an example of these types of software, in which, every action (adding items to the basket, check-out, payment) is associated with its own outputs or log events. The meaning of the error, as an undesired output or log observation, is clearly determined by the input under this assumption [7]. When an erroneous output is detected, software developers investigate the corresponding input to find out where, in the code, it triggers the error. We called this situation *Software-Level Activity (SLA) logging* due to the fact that the output log is a trace of software activities and outputs.

In contrast to the SLA logging, for certain types of software, associating a fault situation to a specific input-output is not explicit. Instead, the internal faults drive the computer system into a period of anomalous behavior, which may end up in a system failure. Many of complex software systems experience similar situation [9] [10]. For instance, a network appliance, a cellphone, micro-service systems , cloud infrastructures or a multi-user operating system may experience a period of anomaly that ends up in a system reboot. In such systems, there is a time epoch between a failure and the input that caused the failure. In this condition, when gathering *SLA* logs and outputs is not feasible, a practical way to find anomalous behavior and their root cause input is system-Level *Monitoring logging*. In *Monitoring logging*, we can record some different metrics that include the host or container CPU usage, memory utilization, and storage capacity. These metrics give a broad understanding of the infrastructure’s status and how well it suits the application’s requirements. However, other types of metrics, such as the slowest and most time-consuming requests, provide a deeper understanding. The *Monitoring logging* has an intuitive property: The rates of input arrivals and status sampling can be different, and generally, the status information is sampled in relatively slower pace than the input arrivals. Therefore, this approach has two serious challenges: the meaning of *error* is not directly linked to a specific input. Thus, we search for anomalous behavior instead of errors. But the second challenge is to relate inputs to the anomalies. In other words, a detected period of anomaly in the system spans over numerous inputs. Hence, finding an input or inputs that caused the anomalous behavior is challenging due to the slow pace of status sampling.

There are two different categories of techniques, one based on machine learning and the other based on graph analysis for Root cause detection. [11] constructs an event causality graph, whose basic nodes are monitoring events such as performance-metric deviation events, status change events, and developer activity events. These events carry detailed information to enable accurate RCA. The events and the causalities between them are constructed using specified rules and heuristics. GROOT constructs a real-time causality graph based on events. Since it construct the causality graph, it is difficult to create graphs. Still, updating the system for changing needs becomes a challenge.

In the ML based categories, several related supervised learning approaches for root cause detection in software systems have been explored in academic research and industry. One approach for root cause analysis with convolutional neural networks and recurrent neural networks in cloud infrastructure, has been proposed in [12]. Furthermore, [13] discuss the importance of feature engineering and model selection in building accurate classifiers, such as k-nearest neighbors, for anomaly detection and root cause analysis in large-scale distributed systems. [14] explores the application ensemble

[1] https://github.com/PHILAE-PROJECT/

methods, for predicting root causes of software failures based on log data and error reports. The study examines the impact of different feature representations and classification algorithms on the performance of root cause prediction models. Although effective in different scenarios, these papers highlight the importance of labeled data for identifying root causes of performance issues and failures which is in general time consuming to gather or even impossible in some cases.

Unsupervised approaches for root cause detection have also been proposed in the past years [15], [16], [17], [18]. These approaches are mostly employed for TSR; Greedy, Clustering, and Search approaches [15], [19]. The existing clustering-based approaches employ supervised clustering algorithms to group test cases. Applying clustering-based approaches has received a deal of attention [19], [20], [17]. Some similarity-based approaches have been proposed for clustering, which generally tries to find similar test cases and remove redundancy [16]. These approaches mainly differ from each other in terms of the type of algorithms used.

However, our proposed approach differs from other unsupervised learning approaches on three main points:

- *Root-Cause event detection*. In this study, our primary focus lies in detecting root causes within incoming events by examining the correlation between these events (such as inputs, network requests, new connections, user logins, etc.) and any unusual software behavior. For example, we aim to identify patterns where a failure occurs consistently after specific remote user login attempts or combinations of events. This initial stage of software testing aims to isolate actions or sequences of actions that collectively lead to abnormal behavior or software failure. While a subsequent phase of source code analysis, conducted by software developers, may follow, this paper concentrates on the initial stage. We refer to this task as root-cause *event* detection to underscore this primary focus, which, to the best of our knowledge, represents the first study on this topic.
- *Application-specific vs generic solutions*: software artifacts differ in many aspects, from their architecture, the nature of the data that they process, their response time, and network activities to their log output information, as well as in the nature of their faults, the complexities of their causality, and how their effects are projected on the log files. And this is only if we want to name a few. Therefore, available approaches are either limited to a specific software environment or they make some basic and general assumptions about how software behaves. This fact limits the generalization of the existing methods in some particular cases. For instance, existing solutions cannot address *status monitoring* log files.

  Our approach is generic and is applied to two different case studies with specific log mining task.
- *Discovering mutual and multi-causation effects*: To the best of our knowledge, none of the existing approaches are meant to find mutual effects of different events to trigger a bug. For instance, in the scanner case study, some anomalies were caused by a group of events in some specific order of appearance. Hence, one gap to fill was to learn about these complex causalities between events and failure or anomaly that our approach can tackle.

3. TRAIL

In this section, we outline the overall workflow of our proposed approach. First, we explore log analysis, then proceed to cluster test events within a learned embedded space. Finally, we cluster log partitions within the space induced by the clusters of test events.

1. Log Analysis

The first step involves log file preparation, where we distinguish between two types of log files pertinent to our case studies: Monitoring logs, which record computer status information at regular time intervals, as was the case for the internet access device case study; and Software-Level Activity (SLA) logs, predominantly capturing software activities and outputs, relevant for the scanner case study. Detailed presentations of both case studies will follow in Section 4.

1) Log Parsing

Different offline and online log parsing techniques have been proposed recently [21], [22], [23]. The task is challenging due to the diverse event templates arising from software complexity. Moreover, the frequent updating of logging statements is a consequence of the regular changes in program functionality. While logs can vary in structure, especially when produced by specialized logging packages, they typically record events in a fixed format (e.g., timestamp, index, pointer to a context). The initial step in this endeavor is extracting log messages into structured data for analysis

In the scanner case study, we employed test suites consisting of sequences of test cases designed to evaluate software programs. Each test case within these suites explores distinct action-input-output combinations, which in turn reveal unique system behaviors. We opted to differentiate between comparable events with varying inputs and outcomes. Thus, we represented each action-input-output as a triplet, denoted by $[a, p, o]$, where a signifies the action, p denotes the input parameter, and o represents the output parameter. This triplet scheme allows for the comprehensive encoding of all operations, inputs, and outputs and is applicable to various software types with differing input and output parameters [7].

In the internet access device case study, log files contained extensive records of inbound events spanning six months, along with details regarding device status or monitoring [9]. Each test log comprised a lengthy list of input events with timestamps, generating a complete day of monitoring for each monitoring record sampled every 5 minutes.

2) Log partitioning

Log partitioning aims to split logs into independent sequences of events, reducing noise in the learning process and enhancing machine learning's ability to discern various event source behaviors.

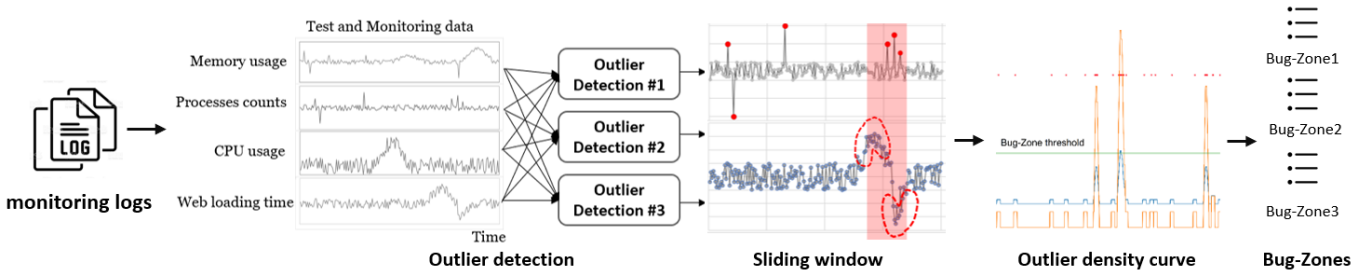

Figure 2: Example of log partitioning in the internet access device case study. It consists of three main calculations: outlier detection, outlier counter sliding window, and outlier density curve with standardization. More details can be found in [6].

In the scanner case study, we partition the logs upon users by segmenting input events into sequences based on the specific "customer ID" attribute.

In the internet access device case study, software-level activity logging such as timestamps and log identifiers often divide logs. Timestamps record event occurrence times, while log identifiers indicate related system actions or message transfers. However, in monitoring logging there is no direct link between input events and logs, with numerous events collectively influencing status information. Thus, correlating individual events to status changes is complex. Status records are not exclusive to specific users or sources, making partitioning challenging. To partition monitoring log files, we utilized the concept of Bug-Zones proposed in [24]. A Bug-Zone signifies a period when a software system exhibits abnormal behavior, disrupting its normal operation and affecting status information. This concept served as an initial exploration into defining faults in monitoring logs, where abnormal internal conditions disrupt system continuity for a duration. For this purpose, we developed a log-partitioning tool, called the Bug-Zone Finder, comprising several steps: anomaly detection, sliding window outlier counting, standardization, outlier density curve generation, and Bug-Zone extraction. Initially, outlier detection functions preprocess monitoring data, employing various techniques that can be compared for consensus. The sliding window counts outliers within specific time frames, with higher values indicating increased outlier occurrences. Standardization adjusts the sliding window output's mean to zero and standard deviation to one, resulting in the Outlier Density Curve (ODC), as defined in [24]. Bug-Zones are identified when the ODC surpasses a predefined threshold. The general workflow of log partitioning is illustrated in Figure 2. The Bug-Zones, derived from the monitoring log, help identify potential causes among input events. We assume the cause(s) of Bug-Zones precede their onset, within a defined timeframe before their impact on status information. Thus, the next step involves extracting input events occurring before the Bug-Zone (Pre-Bug-Zone), based on observations from system developers about outlier density and the duration of root causes preceding Bug-Zones.

2. Event embedding and clustering

Following the log analysis phase, log partitions are initially represented as sequences of events. From these sequences, contextual information is leveraged to derive event embeddings. Recently, various Neural Network-based methodologies, including recurrent neural networks [25], long short-term memory networks [26], and transformer architectures [27], have emerged to learn embedded spaces for sequences. These techniques effectively capture semantic similarities and relationships among sequence elements. However, a significant limitation of these approaches is their high data requirements for effective learning.

To address this challenge, we adopted the word2vec technique [28] a widely-used method in natural language processing. This technique allows for the acquisition of distributed representations of words within continuous vector spaces from textual corpora in an unsupervised manner, rendering it suitable even for smaller datasets. Studies have demonstrated that these learned word embeddings effectively capture semantic relationships and syntactic patterns within documents, rendering them highly relevant to our specific case as well.

In the subsequent step of this phase, events are clustered within the identified embedded space; using X-means, a variant of K-means clustering, which enhances cluster assignments through iterative subdivision, while keeping the most optimal splits until reaching the Bayesian information criterion threshold [29]. Clustering serves to group events exhibiting similar semantic characteristics or relationships, thereby facilitating the identification of patterns and connections within the logs. This process aids in revealing the underlying structure and relationships within the log data.

3. Log partitions representation and clustering

The final phase of our approach involves representing sequences of events as vectors. A common approach for constructing distributed embedding of sequences, irrespective of the sequential order of events, involves averaging the vector representations of each element within the sequence [30].

In our case, we propose generating a vector representation for each log-partition based on identified event clusters found in the previous phase. Suppose that the input events are

partitioned into K clusters; $\boldsymbol{C} = \{C_1, \ldots, C_K\}$. Then, each log-partition ℓ_p, is represented in a vector space of dimension K induced by $\boldsymbol{C}$,

$$\vec{\ell_p} = \left(\frac{n_{C_1}}{\sum_{k=1}^{K} n_{C_k}^2}, \ldots, \frac{n_{C_K}}{\sum_{k=1}^{K} n_{C_k}^2} \right)^\top,$$

where n_{C_k}, indicates the number of events from cluster C_k contained in ℓ_p. Hereafter, we refer to the space formed by the clusters of events used to represent the log-partitions as the "concept space". A similar idea has been proposed for text segmentation [31].

Under this representation, clustering of log-partitions is performed using the X-means clustering algorithm. Log-partitions containing test events of similar frequency will have comparable representations and are assigned to the same cluster. This algorithm explores the space of centroid locations to identify the optimal partition of the log-partitions. This grouping assists in identifying patterns and connections within the data, thereby facilitating the interpretation and comprehension of the underlying structure.

In the case studies, we will elucidate how the proposed method can be applied to find the underlying causal relationship between input events and failures in log files.

4. The case studies

In the following sections, we will provide succinct overviews of the two case studies under consideration.

1. Internet Access Device case study

The internet access device (IAD) case study consists of a large record of incoming events, from Orange Livebox[2] over six months [6]. Concurrently, the device's status and monitoring information were logged during this period. A short description of the two log sets is as follows:

- Monitoring Logs: include a sequence of multivariate samples of the appliance's resource usages like processor, memory, processes, and network. Here is a sample of the monitoring event: "value": 17384.0, "node": "monitoring", "timestamp": "2019-01-14T23:00:18+00:00","domain": "Multi-services", "target": "X1","metric": "stats->mem_cached".
- Test (event) logs: Several clients (PCs) use the internet access appliance to access different services on the Internet, including network activities such as Web surfing, Digital TV, VoIP, Wi-Fi, P2P, Etc. All the clients' requests are recorded on their storage and accumulated later into a large log file on a daily basis (24H). Each log file is a long sequence of input events with their timestamps. Here is an example of a Test log file entry:
  "timestamp": "2018-10-08T08:01:27+00:00", "metric": "loading time", "node": "client03", "target": "http://fr.wikipedia.org", "status": "PASS", "value": 1121.0, .

The challenge of analyzing this case study is more linked to the large difference between the sampling intervals of the monitoring information and the arrival time of the client's requests. While the client requests come in order of a few seconds, the monitoring information is sampled in order of minutes (e.g: 10 min). In other words, in the period between two consecutive monitoring samples, hundreds of test events are recorded in the test logs. Therefore, it is not feasible to directly correlate single input events to changes in the status information, which in turn makes the anomaly's cause detection more complicated.

During the six month of the internet access device case study, certain system failures ensued due to tests introduced into the system during testing. Our objective is to discern sequences of events that are prone to triggering anomalies, while also aiming to condense the size of test records to aid testers in pinpointing critical time periods for anomaly identification. Additionally, we seek to offer intelligent analytics of test execution outcomes, empowering test engineers to prioritize attention on the most error-prone segments of the system under test (SUT).

2. Scanner case study

A barcode scanner is a device used for self-service checkout in supermarkets. The customers (shoppers) scan the barcodes of the items which they aim to buy while putting them in their shopping baskets. The shopping process starts when a customer (client) "unlocks" the scanner device. Then the customer starts to "scan" the items and adds them to his/her basket. Later, customers may decide to "delete" the items. Among the scanned items, there may be barcodes with unknown prices. In this case, the scanner adds them to the basket, and they will be processed later by the cashier, before the payment at checkout. The customer finally refers to the checkout machine for payment. From time to time, the cashier may perform a "control check" by re-scanning the items in the basket. The checkout system then transmits the items list for payment. In case unknown barcodes exist in the list, the cashier controls and resolves them. At the final step, the customer "abandons" the scanner by placing it on the scanner board and finalizes his purchase by paying the bill.

The scanner system has a Java implementation for development and testing and a Web-based graphical simulator for illustration purposes[3]. The web-based version emulates customers' shopping and self-service check-out in a supermarket by a randomized trace generator derived from a Finite-State Machine.

The trace logs of the scanner system contain interleaved actions from different customers who are shopping concurrently. Each customer has a unique session ID that distinguishes his/her traces from another customer.

To artificially inject faults, the source code of the scanner software is mutated with 49 mutants all made by a modification on the source code by hand, as described in [6]. We needed a few logs to be used as the test bench for the proposed method. Hence, we are given three log files with different numbers of

[2] https://en.wikipedia.org/wiki/Orange_Livebox

[3] The simulator can be run at https://fdadeau.github.io/scanette/?simu, traces can be seen in the browser console.

traces: 1K, 100K, and 200K. They include shopping steps for different numbers of clients (sessions): 61, 7078 and 14442 clients, respectively. They were created as random usage logs by a generator of events that simulates the behavior of customers and cashiers. We used mutation testing for evaluation purposes. In the rest of this paper, "session" and client are equivalent.

Mutation testing, a widely adopted technique in academic research, involves modifying the System Under Test (SUT) to simulate potential faults introduced by component programmers, creating what are termed as *mutants*. Testers then develop test cases to reveal these seeded faults. When a test case exposes a discrepancy between a mutant and the original SUT, it is deemed to *kill* or *neutralize* the mutant. The underlying premise of mutation testing asserts that a well-designed test suite can detect all mutants. Testers refine the test suite until it can effectively identify mutants that deviate from the original SUT, as an inadequate test suite may fail to uncover certain mutants. The purpose in scanner case study is to determine the causal relationship between user actions and fault triggering in software. In other words, we want to know which sequence of input events from the user kills individual mutants placed into the source code. In this case study, we are not permitted to re-run the entire trace set in order to determine their influence on software. Instead, we infer the causal relationship between an input event and a software problem solely by observation and learning of their meanings. This condition mimics real-world software testing circumstances in which the number of events in test suites or log files is massive, and running the entire set is prohibitively expensive and time-consuming. Thus, learning the causality effect of event sequences without running them will save time and money. Finally, by running a small set of events and user action sequences, software developers will be able to trigger, chase down, and localize errors in their software.

5. Experimental Setup and Results

In this section, we outline how our proposed method contributes to root-cause event detection from the test event selection and reduction perspective, and present results on the two case studies presented in Section 4.

Test suite reduction plays a central role in enhancing the efficiency of root-cause event detection by streamlining the testing process and focusing on critical areas of the system. By reducing the test suite, the execution of tests related to potential root causes will be prioritized, thereby accelerating the identification and resolution of underlying issues. Moreover, in this context, log files serve as records of past software events, and their minimization proves advantageous for diagnosis and root cause discovery.

This process involves a compromise between the level of test reduction and the coverage of the entire log partition's effect. Put differently, as the number of input events decreases, the likelihood of the minimized set having a lesser fault-triggering effect from the log-partition increases. To maintain equilibrium between the level of minimization and the fault-triggering effect, one can adjust the number of log partition clusters identified in the third stage of our approach, as detailed in Section 3. A higher number of clusters implies a more nuanced distinction between the semantics of the test sequences and a greater representation of test sequences, thereby increasing the likelihood of triggering more failures.

1. Quality Assessment of the Clusters

We utilize the clusters of log-partitions identified in the third stage of our method (Section 3) to discern the correlation between incoming events (such as inputs, network requests, new connections, new user logins, etc.) and any anomalous software behavior or software failure. The initial evaluation focuses on the quality of log-partition clusters induced by the clusters of events (phase 2, Section 3). Since our method operates in an unsupervised manner, we indirectly measure the quality of clusters containing log-partitions in and out of the *Bug-zones* for the internet access device case study. Similarly, for the scanner case study, we measure the quality of clusters containing log-partitions capable of neutralizing less or more than the average number of mutants neutralized by clients, denoted as M_{avg}.

To this end, we employ the *purity* measure, Π, defined as [32]:

$$\Pi(\boldsymbol{C}, \boldsymbol{S}) = \frac{1}{N} \sum_{k} \max_{l} |C_k \cap S_l|, \tag{1}$$

where, N is the total number of log-partitions, $\boldsymbol{C} = \{C_1, \ldots, C_K\}$ is the set of K clusters of log-partitions (phase 3, Section 3). For the internet access device case study, $\boldsymbol{S} = \{S_1, S_2\}$ is the set of log-partitions in (S_1) and out (S_2) of the *Bug-zones*; and for the scanner case study, $\boldsymbol{S} = \{S_1, S_2\}$ is the set of log-partitions capable of neutralizing mutants more (S_1) or less (S_2) than M_{avg}.

Hence, the purity is the sum of the numbers of data points from the most predominant set $S_l \in \boldsymbol{S}$ in each cluster $C_k \in \boldsymbol{C}$ divided by the total number of log-partitions. A higher purity indicates that the clusters are composed predominantly of log-partitions from a single set, which suggests strong separation and distinction between different groups within the dataset.

We begin our evaluation by comparing in Table I, the purity measures of `TRAIL` where log-partitions are found in the concept space and where log-partitions are vectorized by averaging the vectors of the events they encompass before clustering for IAD and scanner case studies. Based on these

TABLE I: Purity measures in percentage (equation 1) of log-partition clusters found in the concept space (Section 3, phase 3) and by averaging the vectors of events for IAD and scanner case studies.

	Concept space	Averaging
Scanner (1K traces)	80%	75%
Scanner (100K traces)	73%	68%
Scanner (200K traces)	72%	71%
Internet Access Device	70%	64%

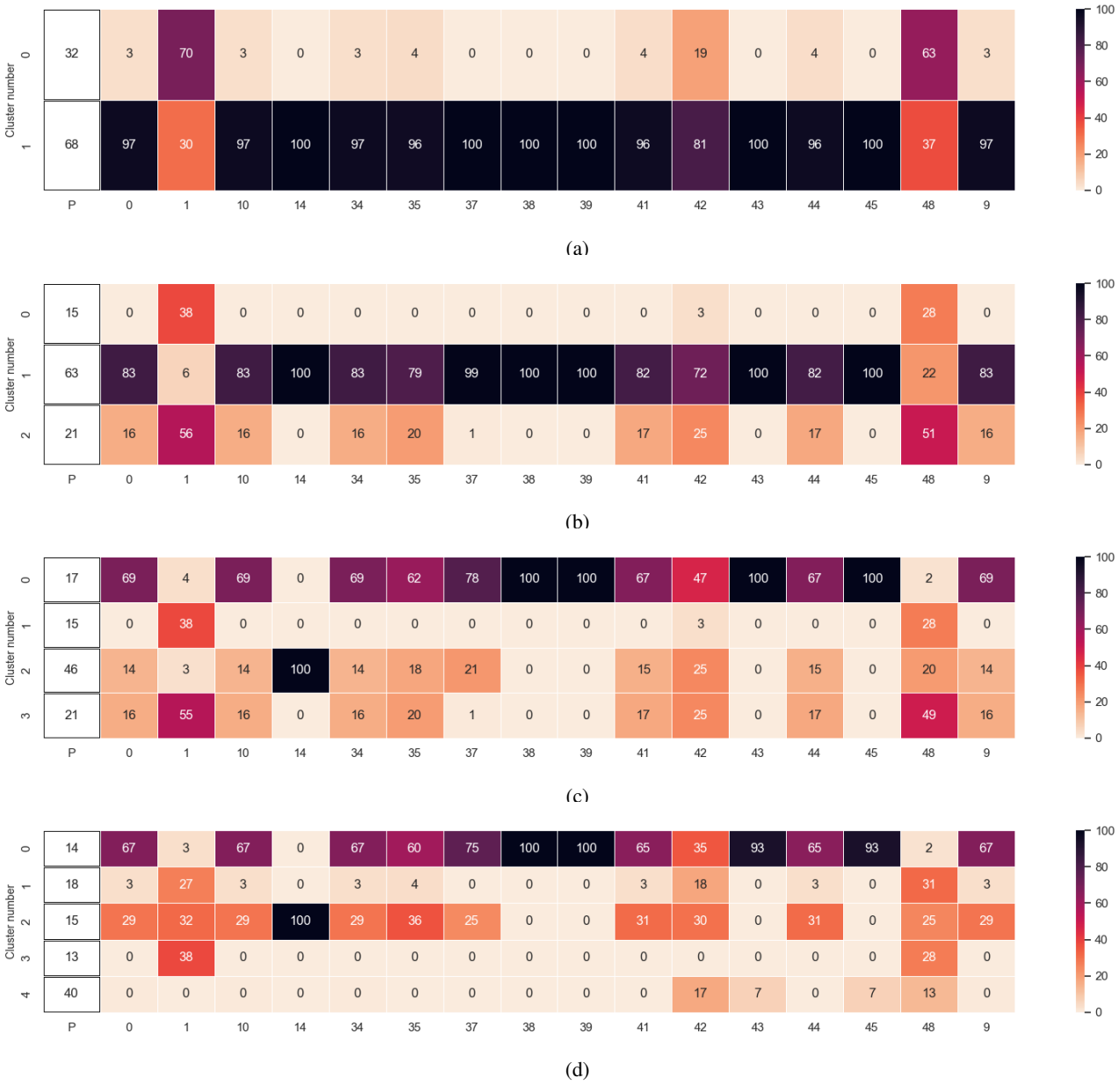

Figure 3: Heatmaps illustrating how TRAIL separates clients with comparable impact on mutant neutralization in two (a), three (b), four (c) and five (d) clusters by clustering log-partitions within the concept space in the scanner case study (100K traces).

findings, it is evident that clusters found in the concept space tend to aggregate more similar log-partitions, in the sense that they contain *Bug-Zones* for IAD case study or they are capable to kill more than M_{avg} mutants for the scanner case study. This superiority over averaging event vectors arises primarily from the independent nature of feature characteristics in the concept space, which correspond to the presence of events in logs that belong to disjoint clusters of test-events (Section 3, phase 3). Conversely, Word2Vec event embeddings are dense embeddings as they map each event to a high dimensional vector space where each dimension represents some latent feature or context of the event [28]. The individual dimensions of these dense embeddings typically are not independent, and when averaging vector characteristics in the embedding space, independence of feature vectors is not generally guaranteed. The results of Table I is in line with previous observations which showed that the performance of K-means, and ultimately the X-means algorithm, can be affected by the scale and correlation of features in the dataset [33, ch. 14], as these algorithms treat each dimension of vectors equally and independently during the clustering process.

Figure 3 shows four heatmaps for three different clusterings

found by TRAIL using the K-means algorithm instead of X-means at phase 3 (Section 3) for varying numbers of clusters $K \in \{2, 3, 4, 5\}$ on the Scanner 100K traces. In this analysis, a total of 13 mutants are neutralized. The horizontal axis represents the mutant numbers, while the vertical axis indicates the cluster numbers. Within each cell, the numerical value signifies the percentage of mutants neutralized by the clients within the corresponding cluster. Consequently, darker shades indicate higher percentages, reflecting a greater efficacy in mutant neutralization by the cluster's clients. The first column, denoted by the label *P*, indicates the cluster size, expressed as a percentage of the total number of clients it encompasses.

In Figure 3a, we note that cluster number 0 identified by TRAIL encompasses 68% of the entire dataset's clients, capable of neutralizing all mutants except three. Conversely, cluster number 1 comprises clients that do not neutralize any mutants except for mutants #1, #42, and #48.

With an increase in the number of clusters, as shown in Figures 3b, 3c and 3d, we continue to observe the partition of clients who do not neutralize any mutants into the same clusters. For example, these clients are predominantly grouped in clusters 1, 3 and 4, constituting 71% of the entire dataset for $K = 5$. While mutant #14 is with the other mutants in Figures 3a and 3b, when we increase K in Figures 3c and 3d, all the clients who neutralize mutant #14 are clustered in a separate cluster (cluster number 2 in both cases).

In summary, the heatmaps in Figure 3 demonstrate that TRAIL effectively separates "fault-maker" clients (those neutralizing mutants) from others, and as the number of clusters rises, it can further distinguish between various fault-maker clients based on the mutants they neutralize. It is worth noting that TRAIL operates in an unsupervised manner and does not require any supervision rendering the proposed method rapid and efficient. While executing the entire dataset took over two days on a high-end PC (Intel vPro i7 with 64GB RAM), the entire TRAIL implementation in Python code runs in less than a minute.

2. Extracting Causality Relations

Using TRAIL, it is also possible to perform test suite reduction in the scanner study case, by identifying a single *representative* client from each cluster that can effectively neutralize as many mutants as the entire cluster. To accomplish this goal, we propose to select clients that are the closest, based on Euclidean distance, to the centroids of the clusters.

Similarly to before, we compare the clustering of log-partitions obtained in the concept space and when the vectors of log-partitions are obtained by averaging the vectors of events they contain.

The outcomes for 1K Traces are illustrated in Table II, employing the Kmeans algorithm in phase 3 with different number of clusters. From these results, it is evident that for different number of clusters $K \in \{7, 8, 9, 10, 11\}$, with our proposed method clients that are closest to the centroids capable of neutralizing more mutants, compared to the Averaging method used for representing log-partitions before clustering. For example, when $K = 8$, clients that are closest to the centroids contain 144 events and have the capability to neutralize 15 mutants. In contrast, employing the averaging method, clients nearest to the centroids contain 137 events but are only capable of neutralizing 8 mutants.

TABLE II: Number of mutants killed and number of events found with our approach and when log-partitions are clustered by averaging the vectors of events.

	TRAIL Concept Space		Averaging	
K	# of mutants killed	# of events	# of mutants killed	# of events
7	12	112	5	106
8	15	144	8	137
9	16	163	8	148
10	16	182	8	158
11	15	196	13	174

Following this, the next step involves assigning the selected clients to software developers, enabling them to examine the content and identify any potential flaws in the software code. We will now present a final analysis focused on root-cause detection that is uniquely achievable by the proposed method. This analysis aims to identify which topic(s) and subsequent actions are responsible for neutralizing mutants, without the necessity of selecting clients and executing them. This is achieved through the representation of log-partitions in the concept space (as described in phase 3, Section 3). Specifically, we accomplish this objective by subtracting the nearest log-partitions from the centroids of two clusters. One cluster encompasses log-partitions capable of neutralizing the highest number of mutants, while the other contains log-partitions that predominantly fail to neutralize any mutants.

For instance, in the case of the 100K traces dataset, by subtracting the nearest log-partitions from the centroids of clusters 0 and 4 (as illustrated in Figure 3d), we derive a vector in the concept space. The most prominent characteristic of this vector corresponds to the dimension that exhibits the greatest differentiation between the two initial vectors, which in this case corresponds to:

Prominent events: ['ajouter', 'Barcode', '0'], ['fermerSession', 'Nothing', '0'], ['payer', 'Price-integer', 'Float Number'], ['payer', 'Price-float', '0']

The actions are shown in triplet format. This analysis tells that adding bar-codes (ajouter), payment (payer) and closing sessions (fermerSession) are the three actions that triggers the most faults in the software. In the context of the Internet Access Device case study, we identified certain network actions, all falling under a fault-maker topic, which were directly associated with the device's problematic areas. A report on the fault-maker actions were submitted to the owner company. For the sake of confidentiality, details on the root-cause analysis of this case-study is protected by the owner company.

6. Conclusion and Perspectives

This paper presents a novel approach for automatically identifying root causes of events across two case studies. Our approach, an unsupervised technique, discerns patterns indicative of potential root causes by clustering log-partitions

within the space induced by event clusters. Consequently, log-partitions containing the same events with similar frequencies tend to be grouped together in the same cluster. This inherent property enables the extraction of meaningful insights from unstructured log messages, facilitating the grouping of log-partitions with common root causes into cohesive clusters.

The results obtained from our case studies validate the practical applicability and effectiveness of our proposed method. Through empirical evaluations conducted on both the Internet Access Device and scanner case studies, we illustrate how our approach significantly enhances root-cause event detection performance. Additionally, our method streamlines the testing process by reducing the test suite. These findings underscore the potential of our approach not only in identifying root causes events but also in understanding the underlying causes of these root causes.

Further studies are needed to compare our purity measures. We want to compute the purity measure in the second way, wherein we obtain the vector of log-partitions by the concept space method. The purity measure in equation 1 only evaluates how well clusters differentiate clients who neutralize more mutants from those who neutralize fewer mutants. However, it does not differentiate clients based on the *type* of mutants they neutralize. In the future, we will consider the type of mutants when we calculate the equation 1.

There are many studies that apply various techniques and algorithms for new test generation such as [34]. We plan to extend our study to have a wider log mining functionality, for instance, apply the proposed method to generate new test cases to automate regression testing. To this aim we need to identify sparse areas in log-partitions and try to generate sequence of test events in this space. In addition, we need to work and get more results in the test selection part. For instance, we have to study the criteria for the selection of representatives from each log partition, instead of selecting the nearest client to the center, we can select the longest session or the most diverse one.

It would also be interesting to extend our approach using learning to rank techniques with partially labeled multi-modal data [35], particularly in the context of test case prioritization and automated test suite optimization; as it is often challenging to determine the order in which test cases should be executed to maximize the likelihood of detecting bugs early. Learning to rank algorithms can learn from historical data, including information about test case outcomes and code changes, to rank the test cases based on their potential to reveal defects. This prioritization can lead to more efficient testing by identifying critical test cases that are likely to uncover issues early in the testing process. Also, as software systems evolve in time, the test suite can become large and redundant, leading to increased testing time and maintenance efforts. Learning to rank methods can be employed to automatically identify and eliminate redundant or ineffective test cases from the test suite.

ACKNOWLEDGMENT

The authors acknowledge the support of the French Agence Nationale de la Recherche, under grant ANR-18-CE25-0013.

REFERENCES

[1] Cagatay Catal and Deepti Mishra. Test case prioritization: a systematic mapping study. *Software Quality Journal*, 21:445–478, 2013.

[2] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R Lyu. A survey on automated log analysis for reliability engineering. *ACM computing surveys (CSUR)*, 54(6):1–37, 2021.

[3] Adam Oliner and Jon Stearley. What supercomputers say: A study of five system logs. DSN '07, page 575–584. IEEE Computer Society, 2007.

[4] Wei Xu, Ling Huang, David Patterson, and Michael Jordan. Mining console logs for Large-Scale system problem detection. In *Third Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SysML08)*, 2008.

[5] Marcello Cinque, Domenico Cotroneo, and Antonio Pecchia. Event logs for the analysis of software failures: A rule-based approach. *Software Engineering, IEEE Transactions on*, 39:806–821, 06 2013.

[6] Bahareh Afshinpour. *Mining Software Logs with Machine Learning Techniques*. Theses, Université Grenoble Alpes [2020-....], September 2023.

[7] Bahareh Afshinpour, Roland Groz, Massih-Reza Amini, Yves Ledru, and Catherine Oriat. Reducing regression test suites using the word2vec natural language processing tool. In *SEED/NLPaSE@ APSEC*, pages 43–53, 2020.

[8] Anunay Amar and Peter C Rigby. Mining historical test logs to predict bugs and localize faults in the test logs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 140–151. IEEE, 2019.

[9] Bahareh Afshinpour, Roland Groz, and Massih-Reza Amini. Telemetry-based software failure prediction by concept-space model creation. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, pages 199–208. IEEE, 2022.

[10] Lingzhi Wang, Nengwen Zhao, Junjie Chen, Pinnong Li, Wenchi Zhang, and Kaixin Sui. Root-cause metric location for microservice systems via log anomaly detection. In *2020 IEEE International Conference on Web Services (ICWS)*, pages 142–150. IEEE, 2020.

[11] Hanzhang Wang, Zhengkai Wu, Huai Jiang, Yichao Huang, Jiamu Wang, Selcuk Kopru, and Tao Xie. Groot: An event-graph-based approach for root cause analysis in industrial settings. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 419–429. IEEE, 2021.

[12] Andrew McDole, Mahmoud Abdelsalam, Maanak Gupta, and Sudip Mittal. Analyzing CNN based behavioural malware detection techniques on cloud iaas. In *Cloud Computing*, pages 64–79, 2020.

[13] Bingming Wang, Shi Ying, Guoli Cheng, Rui Wang, Zhe Yang, and Bo Dong. Log-based anomaly detection with the improved k-nearest neighbor. *International Journal of Software Engineering and Knowledge Engineering*, 30(2):239–262, 2020.

[14] Ran Li, Lijuan Zhou, Shudong Zhang, Hui Liu, Xiangyang Huang, and Zhong Sun. Software defect prediction based on ensemble learning. page 1–6. Association for Computing Machinery, 2019.

[15] Saif Ur Rehman Khan, Sai Peck Lee, Nadeem Javaid, and Wadood Abdul. A systematic review on test suite reduction: Approaches, experiment's quality evaluation, and guidelines. *IEEE Access*, 6:11816–11841, 2018.

[16] Emilio Cruciani, Breno Miranda, Roberto Verdecchia, and Antonia Bertolino. Scalable approaches for test suite reduction. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 419–429. IEEE, 2019.

[17] Carmen Coviello, Simone Romano, Giuseppe Scanniello, Alessandro Marchetto, Giuliano Antoniol, and Anna Corazza. Clustering support for inadequate test suite reduction. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 95–105. IEEE, 2018.

[18] André Reichstaller, Benedikt Eberhardinger, Hella Ponsar, Alexander Knapp, and Wolfgang Reif. Test suite reduction for self-organizing systems: a mutation-based approach. In *Proceedings of the 13th International Workshop on Automation of Software Test*, pages 64–70, 2018.

[19] Frédéric Tamagnan, Fabrice Bouquet, Alexandre Vernotte, and Bruno Legeard. Regression test generation by usage coverage driven clustering on user traces. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 82–89. IEEE, 2023.

[20] Carmen Coviello, Simone Romano, and Giuseppe Scanniello. Poster: Cuter: Clustering-based test suite reduction. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 306–307. IEEE, 2018.

[21] Meiyappan Nagappan and Mladen A Vouk. Abstracting log lines to log event types for mining software system logs. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 114–117. IEEE, 2010.

[22] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. Towards automated log parsing for large-scale log data analysis. *IEEE Transactions on Dependable and Secure Computing*, 15(6):931–944, 2017.

[23] Hetong Dai, Heng Li, Che-Shao Chen, Weiyi Shang, and Tse-Hsun Chen. Logram: Efficient log parsing using n n-gram dictionaries. *IEEE Transactions on Software Engineering*, 48(3):879–892, 2020.

[24] Bahareh Afshinpour, Roland Groz, and Massih-Reza Amini. Correlating test events with monitoring logs for test log reduction and anomaly prediction. In *The 6th International Workshop on Software Faults(IWSF)*. IEEE, 2022.

[25] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306, 03 2020.

[26] Benjamin Lindemann, Benjamin Maschler, Nada Sahlab, and Michael Weyrich. A survey on anomaly detection for technical systems using LSTM networks. *Computers in Industry*, 131:103498, 2021.

[27] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Conference of the North American Chapter of the Association for Computational Linguistics*, pages 4171–4186. Association for Computational Linguistics, 2019.

[28] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[29] Dan Pelleg and Andrew W. Moore. X-means: Extending k-means with efficient estimation of the number of clusters. In 17^{th} *International Conference on Machine Learning*, pages 727–734. Morgan Kaufmann, 2000.

[30] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196, 2014.

[31] Jean-François Pessiot, Young-Min Kim, Massih R Amini, and Patrick Gallinari. Improving document clustering in a learned concept space. *Information processing & management*, 46(2):180–192, 2010.

[32] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[33] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[34] Mark Utting, Bruno Legeard, Frédéric Dadeau, Frédéric Tamagnan, and Fabrice Bouquet. Identifying and generating missing tests using machine learning on execution traces. In *2020 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pages 83–90. IEEE, 2020.

[35] Nicolas Usunier, Massih-Reza Amini, and Cyril Goutte. Multiview semi-supervised learning for ranking multilingual documents. In *Proceedings of the 2011 European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part III*, ECML PKDD'11, page 443–458, Berlin, Heidelberg, 2011. Springer-Verlag.