Telemetry-based Software Failure Prediction by Concept-space Model Creation

Bahareh Afshinpour, Roland Groz, and Massih-Reza Amini
Univ. Grenoble Alpes, CNRS, Grenoble INP, Grenoble, France
{bahareh.afshinpour, Roland.Groz, Massih-Reza.Amini}@univ-grenoble-alpes.fr

***Abstract*—Telemetry data (e.g.: CPU and memory usage) is an essential source of information for a software system that projects the system's health. Anomalies in telemetry data warn system administrators about an imminent failure or deterioration of service quality. However, input events to the system (such as service requests) are the cause of abnormal system behaviour and, thus, anomalous telemetry data. By observing input events, one might predict anomalies even before they appear in telemetry data, thus giving the system administrator even earlier warning before the failure. Finding a correlation between input events and anomalies in telemetry data is challenging in many cases. This paper proposes a machine learning approach to learn the causality correlation between input event sequences and telemetry data. To this aim, a Natural Language Processing(NLP) approach is employed to create a concept space model to distinguish between normal and abnormal test sequences. Based on a vectorized representation of each input sequence, the concept space indicates whether the sequence will cause a system failure. Since the meaning of fault is not established in system status Telemetry-based fault detection, the suggested technique first detects periods of time when a software system status encounters aberrant situations (*Bug-Zones*). An extensive study on a real-world database acquired by a telecommunication operator and an open-source microservice software demonstrates that our approach achieves 71% and 90% accuracy as a Bug-Zones predictor.**

***Keywords*—anomaly prediction, incident diagnosis, log analysis, machine learning, software testing, test automation.**

I. Introduction

Many software systems in operation are monitored by system administrators or supervisors to check whether the system is running correctly and provides the expected service it has been set up for. Thus, apart from the flow of normal inputs and outputs that correspond to the delivery of the functions expected from the system, additional measurements on the software are collected regularly and typically sent to a distinct (and possibly remote) supervision system, hence the name "telemetry" [1] for such measurements. In many systems, all such events are stored in software logs, thus enabling post-production analysis. In this paper, we are dealing with systems where both types of logs are collected and available:

- event logs or *input logs*: that record all inputs (and possibly outputs) that correspond to the functional behaviour of the system
- monitoring logs: that record the series of telemetry measurements

Actually, such logs can also be collected during development, at least when the system is complete, typically for testing activities, such as system or regression testing. And in a DevOps approach, there would often be processes to investigate the logs. In testing or in post-failure analysis, logs are the basic source of information to identify failures or faults and to try to relate them to the events that may have caused them.

There might be a large propagation delay between an internal fault occurrence moment and the moment that its effect appears on the output. Due to this propagation delay, the computer system experiences a period of aberrant behaviour and finally terminates in a system failure. Complex computer systems, such as cell phones, network appliances and distributed operating systems are prone to such behaviour if we want to name only a few. The delay between the fault and the system failure makes it difficult to detect its root cause. Yet, identifying the period of anomaly and finding its root cause is a crucial task for several stakeholders in software systems engineering. First, System administrators who need a predictor to foresee a system failure by observing an aberrant behaviour and second, software testers who are looking into large log files for a failure's root cause to solve a bug in the source code.

In an operational mature software system, failures may scarcely occur during normal operation or even during endurance testing. In this case, analyzing a large sequence of input and output events might be impossible or impractical. An alternative is to leverage the *monitoring logging,* in which the system's *telemetry* or *status* information (such as CPU and memory usage time sequences) is recorded and later will be analyzed to find abnormal behaviour. The analysis of monitoring logs must be an automated job due to the long sequences of data and rare anomalous periods[2]. In this sense, unsupervised machine learning can be deployed to achieve automated anomaly detection and an online system failure prediction.

In this work, we deploy machine learning to create a model from the monitoring logs and present some steps to correlate input events with the detected anomalies in order to foresee the coming system failure.

The proposed method is a scalable ML approach that can adapt with unlimited status features and information sampling rates into various *monitoring logging* applications. It has two phases: Anomaly detection: where a bundle of anomaly detection and outlier detection methods are tied to detect time periods in which the software systems expose anomalous

behaviour. We call them "*Bug-Zones*" and use them in the second phase to extract *important* events and train, classify and correlate the events with the occurrence of Bug-Zones.

We can employ the results in two directions: First, the important events and periods of time are clues to the software developers to investigate the root causes of the system failure. Second, the constructed model from the ML training can be used to construct an *online predictor* which observes the incoming events and triggers an alarm in case of an imminent system failure.

The proposed method was deployed to process logs of network appliances acquired by Orange (telecommunication operator), a partner of our PHILAE project and also an open-source microservice online software, called Train Ticket benchmark [3]. In both cases, the logs were obtained in testing phases with simulated usage (in the case of the telecom application, over several months of intensive usage). Therefore, our paper will often refer to the implications of the approach on test logs. The results are presented in this paper. Based on the work carried out for this project, a tool is published on GitHub [1] repository issued by the ANR PHILAE project.

The rest of this paper is organized as follows. In the following Section, we present our case studies: the Telecom case study that motivated the introduction of the approach, and the Train Ticket benchmark system to assess the applicability on a classical benchmark. Section III describes the abstraction of the problem we are addressing. In Section IV, we explain our proposed method in detail. Section V explains our implementation and empirical results of our case studies. Section VI overviews the work related to our study. We discuss threats to validity in Section VII. Finally, we conclude the paper in Section VIII.

II. The case studies

A. Telecom case study

The first motivation of this research was a telecom internet appliance that provides home internet access. The log suite was a large record of incoming events over six months and the device's status or monitoring information was recorded in the meanwhile. A short description of the two log sets is as follows:

- Monitoring Logs: includes 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","bench": "X3".
- Test (event) logs: Several clients (PCs) use the internet access appliance to access different services on the Internet including network activities such as Web surf, 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. Each log file is a long sequence of input events with their timestamps. Here is an example of the Test log file entry:
  "timestamp": "2018-10-08T08:01:27+00:00", "metric": "loading time", "bench": "XX1", "target": "http://fr.wikipedia.org", "status": "PASS", "value": 1121.0, "node": "client03".

The challenge of analyzing the Telecom 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 the changes in the status information, which in turn makes the anomaly cause detection more complicated.

During the six months of log collection, there are some reboots of the appliance due to either internal faults or intentional resets from the administrators. The manufacturer of the appliance was interested in identifying the cause of system failure among the numerous test events. Moreover, telecom operators would like to know if they can detect and anticipate anomalies in the online system.

B. Train Ticket benchmark

We deployed the proposed approach to another software architecture. This time, we chose an open source microservice software. We studied a widely-used benchmark system for railway ticketing called Train Ticket, which contains around 40 microservices. Train Ticket provides typical train ticket booking functionalities such as ticket reservation, payment, change, and user notification [3]. All the microservices are related to business logic. A detailed description of the system can be found in [4]. Following [3], it is possible to manually inject various kinds of failures, so as to assess whether we can find *Bug-Zone* and predict it based on the test and monitoring logs collected from the benchmark system. In our study, we implemented the injection of one type of failure. Just as in [3], we created a simulated usage of the system (and monitored it) by running Stress-ng in a Docker server. Stress-ng [2] has been designed as a tool to test the ability of a computer system to cope with many types of stress. However, we did not use it for stress testing, but simply as a convenient way of creating simulated traffic for the application. We injected it into the food microservice and recorded CPU and memory usage. In our experiment, we collected a test log in a period of three hours in parallel we recorded the CPU and Memory usage in a monitoring log every five seconds. So, the intervals of the test events are one seconds and the intervals of the monitoring log are five seconds. We replicated 5 status system fault cases in the monitoring log.

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

[2] https://wiki.ubuntu.com/Kernel/Reference/stress-ng

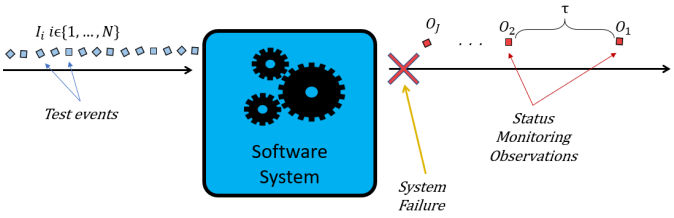


Figure 1. A software system with input and monitoring events

III. Problem Description

We formalize the problem by considering a software system as a function that takes as input a series of events. Examples of such events could be HTTP request, API calls, network packets or database queries. In turn, it produces two types of series: output events (in response to the input events) and status information (the system is monitored for that). In our problem, we are not interested in the detail of the function of the system, so we just abstract the output events by assuming we can observe at some points system failures. The failures could also be observed on the monitoring log. Figure 1 illustrates such a system .

Input events which we call test events in our application context, are denoted by I=[$I_1, \ldots, I_N$], a sequence of N events. In order to be able to predict the imminent arrival of anomalies, we need time information. This is easily ensured by most logging systems in software that record events along with a timestamp. Therefore, a test event I_i is a couple made up of an *event type* which is a member of all possible test events, and a *timestamp* that records when the test event arrives or is executed on the system. On the observation side, the system status is recorded through *monitoring logging*. Observation events O_j are recorded at a lower rate (frequency) than the arrival of input events. So several input events would occur before some O_j happens. O_j records system's status information (e.g.: memory, cpu usage, etc) in an array of values or *metrics*, along with a timestamp. Therefore, a *monitoring* event O_j is a couple consisting of an array of metric values and a timestamp. With our abstraction, the system failures will also be reported (and timestamped) into the monitoring log. In general, we can assume that status sampling is periodic with a period τ (Figure1).

IV. The Proposed Method

The contribution of this paper is a continuation of the work published in [5], in which we defined *Bug-Zone Finder* as an anomaly indicator tool. In this work, we expand it to have a *Bug-Predictor* which will be studied over different representations, model construction scenarios and two case studies. The output is a robust tool to anticipate imminent possibility of a system failure.

A. Bug-Zone *Finder*

The first part of the proposed method makes a use of the *Bug-Zone Finder* presented in [5]. As we presented in the previous work, a *Bug-Zone* is a period of time when the software system exposes an anomalous behaviour. *Bug-Zones Finder* contains these steps: Anomaly Detection, Sliding Window, Standardization and Generating Outlier Density Curve and finally, *Bug-Zone* Extraction.

1) Anomaly Detection: To find these periods, the first step is to deploy outlier detection functions to preprocess the telemetry data. We use two different outlier functions. Each outlier detection function OD_q must accept a multivariate array of monitoring data; it outputs anomalous entries by a Boolean array of outlier records:

$$A_q = OD_q(M) \tag{1}$$

In (1), M= (O_1,…,O_J) is the sampled multivariate monitoring data, in which, each sample O_j contains an array of metric values. A_q, the output of the outlier detection method is an array of size J denoted by A_q= [a_1,…,a_J]. Each a_n is a Boolean value coded by an integer 0 (for false) or 1 (for true) that indicates whether O_j is an anomalous record according to outlier detection OD_q.

2) Sliding Window: As shown in Figure 2, each OD_q gives us one Boolean array A_q. Hence, after deploying outlier detection functions, we have several Boolean arrays with the same size (J). A sliding window can accumulate all Boolean arrays into one array A_{ac}. The sliding window simply counts all "1" or "True" values in all Boolean arrays lying inside a specific window (Figure 3) :

$$A_{ac}[j] = \sum_{\forall A_q} \sum_{k=j-(W/2)+1}^{j+(W/2)} A_q[k] \tag{2}$$

$$j = \{1, \ldots, J\}, A_q[x] = 0 \; for \; x < 1 \; \& \; x > J$$

The sliding window has a size that is denoted by W. A_{ac}[t] is the number of all "1"s in a window by the size of W centered at t. Counting '1' s in the sliding windows must be repeated and accumulated for all the outlier detection output arrays A_q. In Figure 2, we assumed that we have used three outlier detection methods and we have A_1, A_2 and A_3 Boolean outlier arrays. The sliding window outputs higher values when the number of outliers in that period of time increases.

3) Standardization and Generating Outlier Density Curve: The properties of the output of the sliding window, A_{ac}, depends on several factors: number of recorded monitoring features, number of deployed outlier detection functions and the window size. To find *Bug-Zones*, one needs to set a threshold on A_{ac}. To have a constant threshold and simpler design with fewer empirical values, we propose to standardize A_{ac} (the output of the sliding windows). Standardization removes the mean value of A_{ac} and alters its standard deviation to 1. The output is what we call *Outlier Density Curve (ODC)*, from now on. *ODC*=standardization(A_{ac})

4) Bug-Zone *Extraction:* After standardization, *Bug-Zones* are detectable from *ODC*. Bug-Zones are the moments when the outlier density curve rises above the horizontal threshold line (the bottom-right of Figure 2).

Each Bug-Zone is a pair of timestamps of the beginning and the ending events of the Bug-Zone, denoted by $BZ \rightarrow T_B$ and $BZ \rightarrow T_E$.

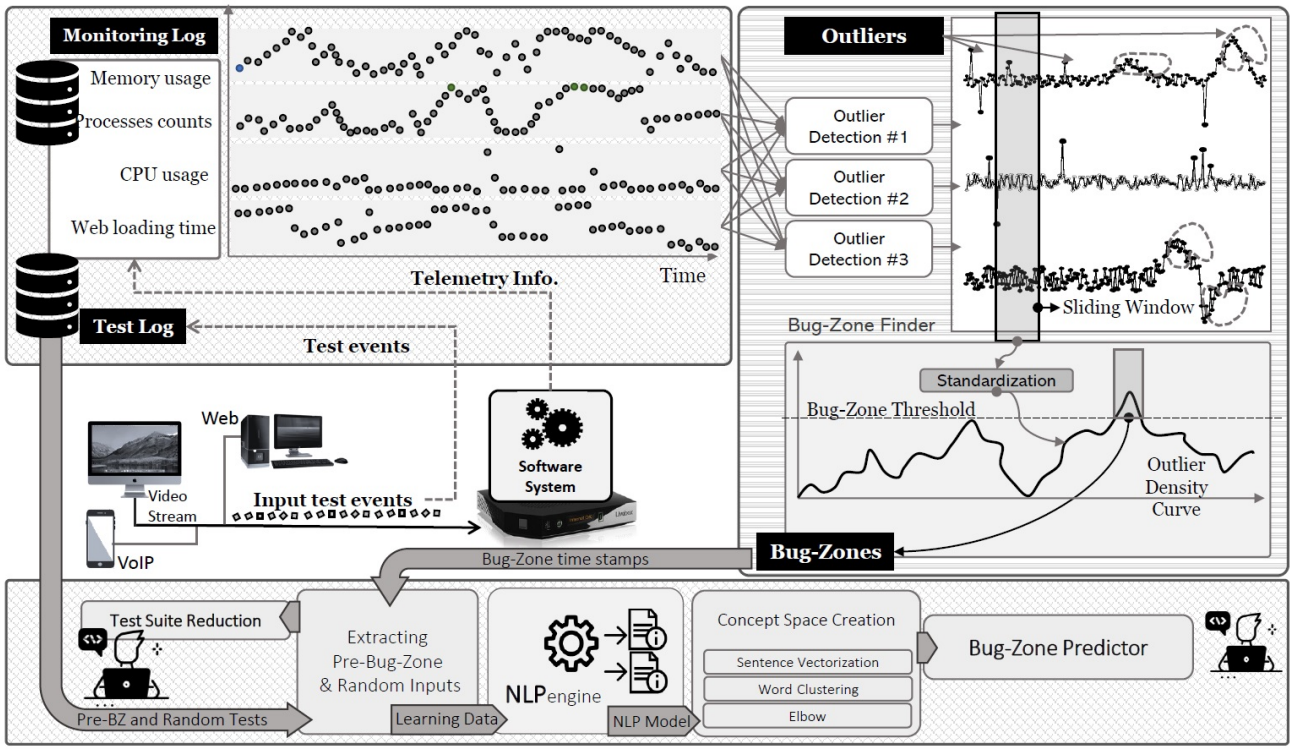


Figure 2. An overview of the proposed method.

B. Learning Phase

The learning phase has three steps:

- Test event extraction
- Model construction
- Sequence representation by concept space creation

Each step will be covered in the following subsections.

1) Test event extraction: At this step, one needs to extract test events in a time range before the *Bug-Zone* (Pre-Bug-Zone). But we will also need to have some non Pre-Bug-Zone inputs to compare with the Pre-Bug-Zone inputs. This can be done by extracting random-time intervals from time ranges outside the Pre-Bug-Zone periods.

The input extraction time range depends on the observations that system developers make on the outlier density curve, considering the root cause may happen how long before the *Bug-Zone*. In our case, we extract test events in a range of 3τ before the center of the *Bug-Zone* ($\frac{BZ_i \rightarrow T_B + BZ_i \rightarrow T_E}{2}$), where τ is the sampling period of the monitoring log (Figure 1). This range proved to exhibit the best results in our case, where sampling is done at a relatively low rate; it can be adapted to other rates of monitoring sampling w.r.t the flow of input events.

Likewise, by creating random timestamps and verifying that they don't fall in the Pre-Bug-Zone periods, we would have a set of random test sequences (*Random-Zones*):

$$PreBZ = \{PreBZ_1, \ldots, PreBZ_Z\} \quad (3)$$

$$PreBZ_z = [I_{z1}, \ldots, I_{zP}] \quad (4)$$

$$Rand = \{RND_1, \ldots, RND_Z\} \quad (5)$$

$$RND_z = [I_{z1}, \ldots, I_{zR}] \quad (6)$$

In (4) and (6), I_{zP} and I_{zR} are test inputs in the designated Pre-Bug-Zone or Random-Zone sets. The number of the Random-Zone sequences is equal to the number of the *Bug-Zones* in order to have a balanced training set. The size of Random-Zone periods was equally chosen to be 3τ.

2) Model construction: At this stage, the extracted Pre-Bug-Zone test events are used to construct a model. Each Random-Zone or Pre-Bug-Zone input array is treated as a sequence. Likewise, each test event in that array is treated as a one-hot-coding vector. We employed a contextual sequence model proposed by [6] to learn the representation of each test event. The model maps then each type of test events into a vector. The array size is $|\phi|$, in which, ϕ is a set of all possible test event types, called *vocabulary*. Likewise, the dimension of each vector in the array is $|\phi|$.

$$NLPModel = NLPEngine(PBZ, Rand) \quad (7)$$

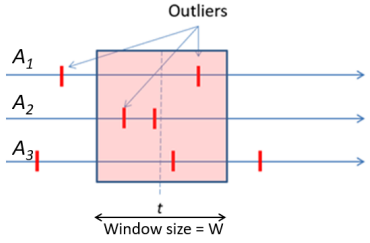


Figure 3. A sliding windows over anomaly detection arrays

$$NLPModel = \{V_1, \ldots, V_{|\phi|}\} \quad (8)$$
$$V_i = [f_1, \ldots, f_{|\phi|}], f_i \epsilon R \quad (9)$$

Each $NLPModel$ output vector V represents a word, and in our case, a word is an input event type E. The distance between two vectors determines how two words (input events) are semantically close. From now on, we interchangeably use "word" term for an NLP vector representing an input event type and "sentence" term for an array of vectors representing an array of input events (e.g. a Pre-Bug-Zone input events).

3) Sequence representation by concept space creation: The created model gives vectors that represent the test events in the *vocabulary*. Therefore, a Pre-Bug-Zone test array $PreBZ_z$ or Random-Zone test array $Rand_z$ could be represented by an array of vectors (a sequence) denoted by $Rand_z^V = [I_{z1}^V, \ldots, I_{zP}^V]$ and $PreBZ_z^V = [I_{z1}^V, \ldots, I_{zR}^V]$.

The representation above is an array of vectors. To create a single-vector representation for each sequence, we need to combine all the vectors of a sequence in a way that effectively reflects the semantics of the sequence. Conventionally, to create a vector from an array of vectors, simple *averaging* the array has been the most straightforward way to go [7]. In contrast, our model creates a concept space from the test events by clustering them into groups of similar events and referring to each group as a concept based on a similar idea expressed in [8]. Then, sequences of events are mapped in the space induced by these clusters. The efficiency of simple averaging and concept space representations will be compared in an example in the Train Ticket benchmark subsection.

After creating the concepts, it is possible to determine the conceptual presentation of a sequence by observing its events and the concepts to which they belong. Hence, a Pre-Bug-Zone sequence $PreBZ_z^V$ is represented by a vector of C dimensions:

$$PreBZ_z^{Concept} = [con_{z1}, \ldots, con_{zc}] \quad (10)$$

In which, con_{zc} indicates how many events from a concept $Concept_c$ exist in the Pre-Bug-Zone sequence $PreBZ_z$. Random sequences of events that are not in the Bug-zones are represented in the same manner $Rand_z^{Concept}$.

C. Creating Universal Prediction Clusters

The final step of the learning phase is constructing the Universal Prediction Clusters (UPC) from the sentences. These clusters are essential to differentiate among Random-Zone and Pre-Bug-Zone sentences for the prediction goal. They project all possible topics in the input event sentences. Each input sequence should belong to one of these universal clusters. The distance between an input sentence and UPCs is used to predict a *Bug-Zone*. This will be covered in the following subsections. A clustering algorithm must be employed to create the universal prediction clusters from all $PreBZ^{Concept}$ and $Rand^{Concept}$ sentences. It must return a set of clusters $UPC = \{upc_1, \ldots, upc_U\}$, each of which is designated by its center and its label. The center is simply an element-wise average of the cluster members, and the label is the same as the cluster members in the majority (either Random-Zone or Pre-Bug-Zone,):

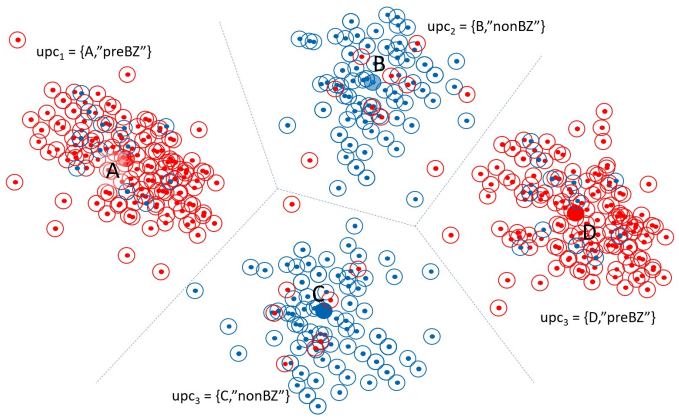


Figure 4. Universal Prediction Clusters (UPC) - Each dot is a sentence processed in the learning phase (Red: Pre-Bug-Zone, Blue: Random-Zone)

$$UPC = \{upc_1, \ldots, upc_U\} = Cluster(PreBZ^{Concept}, Rand^{Concept}) \quad (11)$$
$$upc_i = (center_i, label_i), \quad label_i \epsilon \{"nonBZ", "preBZ"\} \quad (12)$$

Figure 4 illustrates an abstracted example drawn in two dimensions. There are four UPCs, each of which has either of Pre-Bug-Zone (red) or Random-Zone (blue) members in the majority. The cluster label is the same as the label of the majority members. Based on this abstraction image, we describe the Bug-Zone prediction functionalities.

D. Online ML-based Bug-Zone Prediction

Online Bug-Zone prediction gives an advance warning to system administrators about imminent anomalies and probable system failure. To have an online predictor, we can simply train a classifier with the $PreBZ_z^{Concept}$ and $Rand_z^{Concept}$ sets. The classifier learns the classes of sequences that are likely to be Pre-Bug-Zone and distinguishes them from the normal (Random-Zone) sequence. But we propose a second approach to use the created UPCs centers as indicators to determine if a sequence of events may cause Bug-Zone.

To this aim, assume that the latest events (e.g: happened in the period of 3τ, in which τ = the monitoring sampling period) is denoted by $LastInputs = \{I_1, \ldots, I_{3\tau}\}$. The conceptual vectorized version of the latest input events can be calculated from the constructed model in the learning phase: $LastInput^{Concept}$=$[con_{i1}, \ldots, con_{iC}]$. It must be noted that the output vector has a dimension of C, regardless of the number of input events. The closest UPC to this vector determines the prediction verdict. For instance, we imagine that the smallest cosine distance is between $LastInput^{Concept}$ and a Pre-Bug-Zone UPC. Therefore, we predict a Bug-Zone to happen soon. By a new input arrival, updating $LastInput^{Concept}$=$[con_{i1}, \ldots, con_{iC}]$ is not a complex task. Assume that an input event $I_{(3\tau+1)}$ arrives and I_1(the oldest event) must be excluded from the calculations. Then, based on the concepts, to which I_1 and $I_{(3\tau+1)}$ belong, one concept

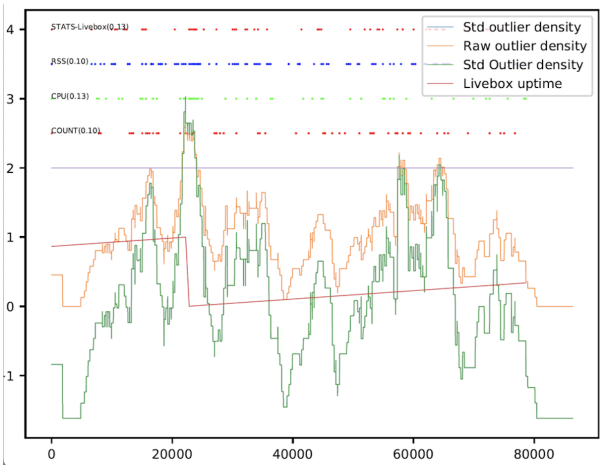


Figure 5. Outlier density curve and detected *Bug-Zones* in the Telecom case study

value in $LastInput^{Concept}$ must be decremented, and another one must be increased. The cosine distance must be calculated again to find the closest UPC.

V. Implementation and Evaluation Results

In this section, we evaluate the effectiveness of our method. We target the following research questions:

$Q1$: Can our model distinguish Pre-Bug-Zone from Random-Zone sequences accurately enough ?

$Q2$: How effective is the proposed approach in predicting *Bug-Zones*?

$Q3$: What is the complexity of the proposed approach?

A. Experimental Setup

We used standard library implementation of classical ML methods, and orchestrated the steps of the approach by developing a Python 3.x script. The first step is based on outlier detection. We experimented two outlier detection methods, Local Outlier Factor and Isolation Forest [9], [10]. These two algorithms belong to the unsupervised outlier detection method and play an important role as anomaly detection methods. Isolation Forest is more efficient and more stable than the LOF. However, the Isolation Forest shows some shortcomings in some experiments. Many normal samples will affect the ability to isolate abnormal points when there are a large number of samples [11].

1) Telecom case study: In that case study, each monitoring event is a collection of metrics. So we processed the multivariate information to identify outlier entries. Each log file corresponds to a full day of monitoring, with samples taken on 5-min periods. Therefore, we associate an array of 288 multivariate samples to each log (288×5 min = 24 hours). In the illustration, we only show 5 metrics, but in reality each sample contains 26 metrics.

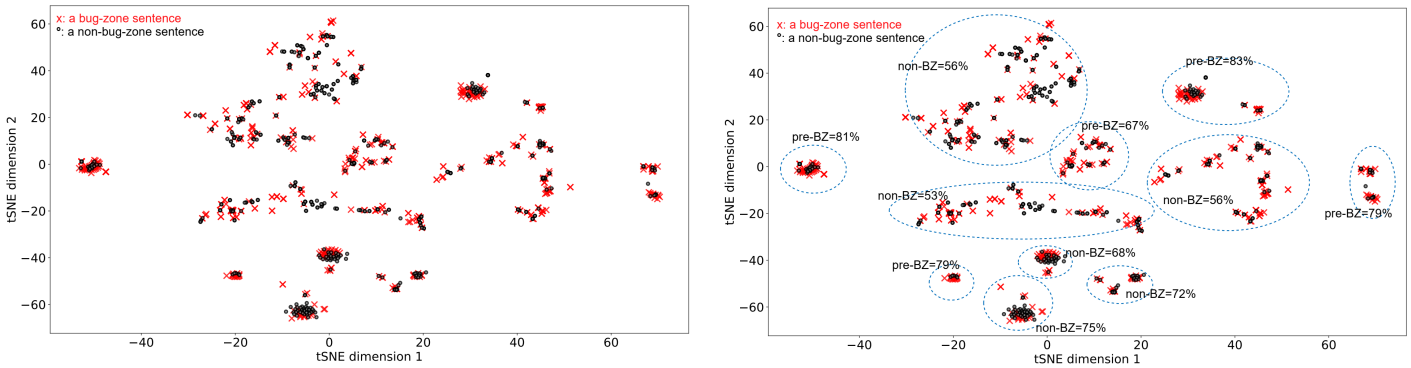
Noticeably, we found how the two outlier detection methods complement each other. Actually, we could add more outlier detection methods in the first step, so as to accumulate all their detection strengths.

Figure 5 illustrates the outlier density curve after applying the sliding windows and standardization steps. Outliers detected by LOF and IF methods are represented as scattered dots in the upper part of the figure. Each row of dots belongs to one of the multivariate series of status monitoring. In the middle of the figure, we record the uptime curve of the system (which is one of the recorded metrics): a drop in the line corresponds to a reboot. The upper curve with variations (drawn in yellow) shows the outlier density before standardization, and the lower curve (drawn in green) shows the same after standardization. The threshold for deciding on a Bug-Zone is represented by the horizontal line, which was set at a level of 2. We can see that the green curve overshoots the threshold around the reboot events.

Although the correlation between reboots and Bug-zones is high, it is not 100%. Actually 70% of the reboots are to be found inside Bug-Zones. In fact, not all reboots are fault-related. They might be triggered by power or network failure, which would not be liable to our analysis based on 5-minute sampling. And we also know that reboots are actually triggered by the test team, from time to time, to restart test sessions (and the the proportion is in line with our observations). The high correlation observed, which lies between 70% and 100% (although absence of further data prevented us from computing a more accurate value) indicates that the Bug-Zone finder is effective in finding anomalous behaviour and predicting system failures through status monitoring. Some other detected Bug-Zones were not near a reboot. Therefore, they may come from transient periods of anomalous behaviour ended without a total system failure.

Once we had identified *Bug-Zones*, we were able to extract the Pre-Bug-Zones from the input log, and to choose Random-Zone sequences lying outside the Bug-Zones and Pre-Bug-Zones. The input log sequences combine 175 different elementary test events. Those events become vocabs for our NLP based approach. We were able to identify 589 Pre-Bug-Zone sequences, and picked 568 Random-Zone sequences. In order to create the NLP Model, we implemented word embedding techniques. The 175 vocabs do not correspond to a real complexity in dimensionality, so we first use K-means in combination with Elbow method [12], to create 20 concepts from the 175 event types. Finally, the sequences for Pre-Bug-Zones and Random-Zones are converted to their corresponding concept-space vectors. The prediction is computed in the space of these vectors. Figure 6(A) presents these sequence vectors in 2D space. Each red cross represents a Pre-Bug-Zone test sequence. Random-Zone test sequences are represented by dark dots. We can see that some of the clusters are more clearly associated to a single type of zone: the majority of items (dots and crosses) are either red or dark. There are some mixed clusters with no clear majority. The figures are drawn in 2D, but actually those clusters may not be mixed in higher dimensions. This can be evaluated by a classifier.

In the next step, we clustered, concept space vectors by using K-means clustering method. We used Elbow method [12] to find the number of clusters. The plotted image of



(A) (B)

Figure 6. (A) A projection of Pre-Bug-Zones and Random-Zones vectors in 2D from the telecom case study,(B) UPCs created from telecom case study

the UPCs in telecom case study are depicted in Figure 6(B). In this figure, we can distinguish some UPC clusters in which either red crosses or dark dots are in the majority. The more imbalanced colors in each UPC are, the more meaningful the cluster is. For instance, upc1 has a very high number of Pre-Bug-Zone (red) cross densely located around a circle. This cluster is a vivid Pre-Bug-Zone cluster, since the probability of a cross being Pre-Bug-Zone inside this cluster is relatively high. There are a few balanced UPCs, ups2 and upc8 which both Random-Zone and Pre-Bug-Zone sentences appear almost equally in these clusters. For the sentences that fall inside these UPCs, the prediction won't be accurate. It must be noted that the red crosses are plotted after the dark dots. Hence, there are some dark dots covered by the red crosses that are not observable by the eyes.

2) Train Ticket benchmark: We studied the datasets in the docker container version 20.10.8 deployed on a GPU server with 125 GB memory. As mentioned in the previous section, the tester triggered the injected bug five times during the test period of three hours. Consequently, the anomaly detection engines detected several anomalous values around the bug events, as depicted in Figure 7 by red vertical bars.

The outlier density curve is depicted in Figure 8. By *Bug-Zone* threshold of 1, the threshold line (in green) intersects the outlier density curve in seven Bug-Zones during the testing period. In Figure 8, the red dots are the detected outliers and the blue and orange curves are the raw and standardized outlier density curves, respectively. While five detected Bug-Zones correspond to the bug events, there are two false positive Bug-Zones, which form a total of seven *Bug-Zones*.

Afterward, referring to the test log, seven Pre-Bug-Zone test sequences (sentences) were extracted from the test log file, as well as fifty Random-Zone test sequences outside the *Bug-Zone* periods. We created a word embedding model from the extracted test sequences and used the averaging and the concept space methods to create a vector representation from the extracted test sequences. Figure 9-A shows a projection of the averaged sequence vectors in 2D space, while Figure 9-B shows the same for the concept space representation. In each Figure, red and gray dots represent the Pre-Bug-Zones and Random-Zones test sequences. The two sentence vectorization methods created two main clusters (surrounded by circles) with distinguishable distances on 2D. In both figures, one cluster has five Pre-Bug-Zones test sequences, and the remaining two fall into the other cluster. After the investigation, we observed that the two red dots located among the gray dots in the gray circle are the *Bug-Zones* which were NOT caused by the bug (caused by anomalies of the other tests). Therefore, their semantics are close to the random test sequences. Consequently, we can state that the sentence representation step can correct the false positives

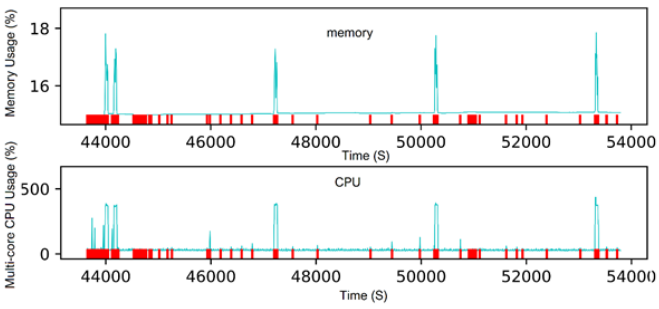


Figure 7. CPU and memory usage during the test period and five bug events in the Train Ticket case study. Detected anomalies depicted by red bars.

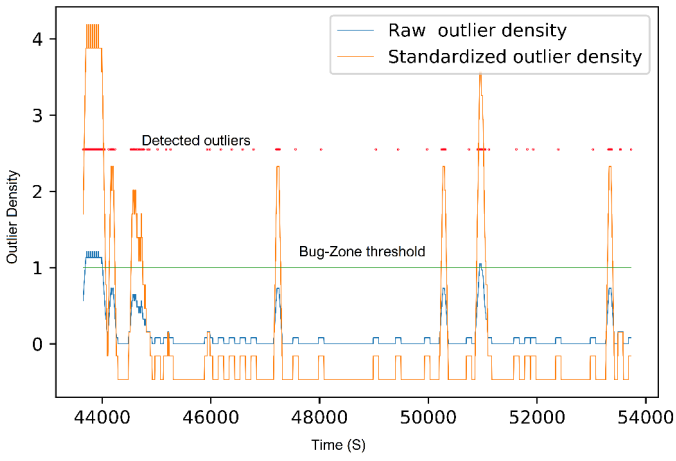


Figure 8. Outlier density curve for the Train Ticket benchmark. The detected outlier shows in red dots and the threshold line in green.

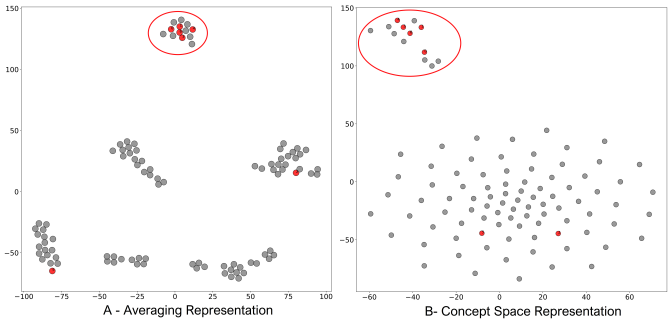


Figure 9. Averaging method vs Concept space method to sentence embedding in the Train Ticket case study

on the Bug-Zone finder step. Also, this observation shows how effectively the proposed method can distinguish between the Bug-Zones based on their causes. Hence, the clusters meaningfully differentiate among different root causes of Bug-Zones.

All red dots must be separated from the gray ones in a circle. Therefore, the gray dots in the wrong circles can base on an estimation of the false positive rate of Bug-Zone prediction. For instance, in Figure 9-B, 8 gray dots are in the red circle (false positive), forming 16% of the population. We expect a similar false positive rate from the Bug-Zone Predictor. One noticeable difference in the concept space figure (Figure 9-B) is the uniformity of the dots in the more significant cluster and its clear distance from the smaller cluster. Apparently, the concept space method better illustrates the two different semantics of the test sequences (Pre-Bug-Zones and Random-Zones). Finally, a Bug-Zone Predictor based on a Random Forest predictor gave us 90.7% accuracy in predicting *Bug-Zones*. A higher precision was expected in this case study due to the lower noise and complexity of the system.

B. Q1: Can our model distinguish Pre-Bug-Zone from Random-Zone sequences accurately enough ?

In our previous study [5], we employed three common classifiers: Support Vector Machines (SVM with radial basis function as the kernel function), Random Forest (RF) and Multi-Layer Perceptron (MLP) from the Scikit-learn library implementations. All three approaches belong to the category of supervised algorithms.

AUC metric or Area Under Curve value [13] is one of the most important metrics for evaluation any classification models performance. It tells how much the model is capable of distinguishing between classes. The AUC metric is a value between 0 and 1. The worst AUC value of a classifier is 0.5 which means the model is not capable of separating the classes. Hence, values near 1 or 0 are desirable. The results show that the Random Forest classifier outperforms the other ones, since the AUC value for Random Forest is 0.75, while the AUC value for SVM and MLP classifier is 0.62 and 0.64, respectively.

TABLE I
AUC VALUES IN DIFFERENT DATASET

Dataset	RF Prediction	UPC Prediction
Telecom	71%	65%
Train Ticket	90.7%	95%

C. Q2: How effective is the proposed approach in predicting Bug-Zones*?*

The effectiveness of a predictor is generally evaluated by its accuracy, and some metrics based on the ratio of its false negatives and false positives. Namely, *Precision*, *recall* and *F1-score*.

1) Accuracy: To train the Bug-Zone predictor, we randomly divided our concept-space dataset (both Pre-Bug-Zone and Random-Zone sequences) into 80% and 20% to train and test the predictor, ensuring each set contains both label 0 (Pre-Bug-Zone) and 1 (Random-Zone). We repeated the random splitting process 30 times for cross-validation and took the average as the results. We chose Random Forest for prediction as a baseline in the previous study, since it was the most accurate among the other classifier. Random Forest, after training, succeeded in correctly classifying 71% of the test dataset in telecom case study and 90.7% accuracy in predicting *Bug-Zones* in Train Ticket dataset. These results imply that it can be used to predict *Bug-Zones* based on a real-time incoming test data. A higher value was expected in Train Ticket benchmark due to the lower noise and complexity of the system.
Table I shows these results. By using UPC prediction method, we achieved 65% and 95% accuracy in Telecom and Train Ticket case studies respectively.

2) Precision, recall and F1-score: We computed common classification metrics, namely, *precision*, *recall*, and *F1-score* which are routinely used in similar work [14] [15] [16] [17] for analyzing accuracy. *Precision* and *Recall* can be formally defined as follows where TP, FP, FN are the number of true positives, false positives, false negatives, respectively : *Precision*=$(\frac{TP}{TP+FP})$, *Recall*=$(\frac{TP}{TP+FN})$. *Precision* is the percentage of correctly predicted Bug-Zones (True-Positive) over all Bug-Zone prediction (True-Positive+False-Positive). It can be considered as the measure of the exactness or correctness of a classifier. A low precision value indicates a large number of false positives [18]. *Recall* is the percentage of Bug-Zones that are correctly predicted in advance among all the Bug-Zones (True-Positive+False-Negative). We can call *Recall* as the measure of the completeness of a classifier. A low *Recall* value indicates many false negatives [18]. As presented in [16], F1-score $(\frac{2*TP}{2*TP+FN+FP})$ is the most used singleton metric, and it is the harmonic mean of precision and recall. From a

TABLE II
PERFORMANCE OF BUG-ZONE PREDICTION ON TELECOM AND TRAIN TICKET CASE STUDIES BY USING RANDOM FOREST PREDICTION METHOD

Dataset	Method	Precision	Recall	F1-score
Telecom	RF classifier	0.68	0.70	0.69
Telecom	UPC Prediction	0.63	0.62	0.61
Train Ticket	RF classifier	0.93	0.93	0.93
Train Ticket	UPC Prediction	0.97	0.75	0.81

tester's point of view, *Recall* metric might be more important, since a lower False-Negative rate (or higher True-Positive) indicates that we are not missing *Bug-Zones* information. For a Bug-Zone predictor, both *Recall* and *Precision* are of interest. A lower *Recall* value indicates that we are missing more *Bug-Zones* and is linked to the cost and consequence of it. But a lower *Precision* value means that we are having more False-Positives, which in turn causes losing confidence in the system, especially when the system takes an automatic measure on a prediction that should not be taken.

3) Comparative Analysis: Table II shows the precision, recall, and F1-score on the telecom and Ticket Train case studies by using Random Forest classifier as a baseline and UPC prediction as a proposed method in this paper. It shows that UPC prediction can achieve to higher precision values (e.g. 97% for Ticket Train). On the other hand, it has lower recall values therefore, higher false negative ratio.

D. Q3: What is the complexity of the proposed approach?

We can distinguish the complexities of the learning phase from the online prediction phase. The learning phase is less critical since it is off-line and needs to be created only once before online prediction. On the other hand, the online prediction is the part that repeats by arriving each time that a new input arrives. On the off-line phase, the complexity of the Bug-Zone finder part is bounded by the outlier detection algorithms, in which the local outlier factor algorithm has the highest complexity in the order of $\mathcal{O}(J^2)$, and J is the number of monitoring samples. Likewise, the complexity of the model creation step is $\mathcal{O}(N.logV)$ where N is the number of test events in the Pre-Bug-Zone and Random sequences, and V is the vocabulary size. For the online Prediction phase, the complexity is $\mathcal{O}(U)$, where U is the number of universal prediction centers. Alternatively, if we use random forest instead of UPCs, the complexity is $\mathcal{O}(T \times D)$, where T is the size of RF and D is the maximum depth. Both parameters depend on the number of test sequences in the learning phase and are expected to be considerably larger than the number of UPCs. Therefore, using UPCs is preferable due to lower complexity in real-time systems.

VI. Related Work

Automated log analysis has received a great deal of interest since it is faster, less costly and more effective than manual log analysis. However, our effort failed to find related works with close assumptions to that of this paper with the aim to be compared against the proposed method. Hence, one cannot adopt them to solve the challenges introduced in the case studies. For example, model extraction methods from log files are not applicable in the monitoring logging domain due to the different nature of the log outputs. Authors of [19] present an approach to automate log file analysis and root cause detection by creating a finite state automaton (FSA) model from successful test sessions and comparing the developed model against failed test sessions. This method and other similar methods would not be effective on status monitoring logs. Due to the huge number of events and their possible combinations before each status record, the created model will be significant and complex. However, FSA and similar workflow abstraction methods are shown to offer limited advantages for complex models [20].

Furthermore, in the majority of approaches, the definition of fault is apparent [2] [21], while in our case, abnormal behaviour of the software artifact is the only lead to diagnosing the system's internal unhealthy condition. Accordingly, supervised approaches employed to analyze these software logs based on their fail and pass labels, are not useful in our case.

Among limited published research on status monitoring logs, authors of [1] find a relation between system events and the changes in monitoring metrics by using statistical correlation methods. However, the approach is limited to incident diagnosis and how a single event affects monitoring metrics. Applying machine learning helps to promote simple and single-event diagnosis to mining events-metrics correlation and have fault detection and prediction.

Different from other works, this paper is one of the few works that exploit system status monitoring observation for bug detection and prediction in software testing. The research is applicable to logs that can come from long test runs on mature software systems, or production logs. The goal of this research was motivated by a telecommunication case study, in which glass-box testing of the embedded third-party software of a network appliance was neither possible nor indeed desirable as it was supposed to have been carried out by the software developers; and the software was mature enough to exhibit faults only in the long run. The implementation of the proposed method can be applied to many similar cases, either in testing or production.

VII. Threats To Validity

The approach has only been validated on two case studies, with different contexts and scales. Obviously, it should be assessed on more case studies. It would have been nice to have a benchmark of representative case studies for fault detection and prediction from monitoring logs, but we did not find one that could correspond to our assumptions and context. The Train Ticket benchmark was inspired from a previous study [22] that already used it to that end; but it is smaller and less complex than most of the surveyed real systems. Also, in that case, we dealt with a simple type of fault. We would need to inject other types of faults in the future. In the case of the Telecom case study, due to human resources issues, we were not able to get complete feedback from the test team to assess the relevance of the detection and of the prediction. Assessing the quality of the prediction would require using the system in a real operational context. Another threat to the validity of our study is the missing relevant papers. We searched the digital libraries that are most likely to cover monitoring log analysis. However, we can not rule out the possibility that we may have missed some relevant studies.

VIII. Conclusion

System status information can be exploited for software testing to find the root cause of system failures and predict them in an online system. This paper presented the Bug-Zone finder and Bug-Zone predictor, two approaches for detecting and predicting anomalous periods in a software system. First, by using different anomaly detection methods, the Bug-Zone finder detects anomalous periods, enabling testers to only focus on the test events near the *Bug-Zones*. Thus, this reduces the testers' efforts and provides valuable information on the events and their causes. Second, by using an ML technique to create a conceptual model from the semantics of the test sequences, the online predictive model enables us to identify sequences of tests that lead to a system failure. Thus, it helps system administrators to foresee system failures in the future. The effectiveness of the two proposed methods was evaluated in two case studies, one from the Orange company and the second one is Train Ticket, an open-source benchmark microservice system. The detected *Bug-Zones* cover at least 70% of the systems reboots (failures) in the telecom case study; Random Forest predictor, after training, succeeded in correctly classifying 71% of the test dataset in the telecom case study and 90.7% accuracy in predicting *Bug-Zones* in Train Ticket dataset. By using the UPC prediction method, we achieved 65% and 95% accuracy in Telecom and Ticket Train, respectively. A higher value was expected in the Train Ticket benchmark due to the lower noise and complexity of the system. These results imply that our created model can be used to predict *Bug-Zones* based on real-time incoming test data by using RF or UPC predictor.

For the continuation of this work, we aim to employ the created concept space and UPCs in finding the root causes of anomalies among the input test events. The achievements of this future work can help software developers to localize and find the cause of system bugs.

Acknowledgment

This work was supported by the French National Research Agency: PHILAE project (N° ANR-18-CE25-0013). The authors are very grateful to Benoit Parreaux for providing a wealth of data on the case study, as well as much advice on the problem statement. We are also thankful to Yves Ledru for his review and helpful discussions.

References

[1] Chen Luo, Jian-Guang Lou, Qingwei Lin, Qiang Fu, Rui Ding, Dongmei Zhang, and Zhe Wang. Correlating events with time series for incident diagnosis. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1583–1592, 2014.

[2] Cheolmin Kim, Veena B Mendiratta, and Marina Thottan. Unsupervised anomaly detection and root cause analysis in mobile networks. In *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*, pages 176–183. IEEE, 2020.

[3] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 47(2):243–260, 2018.

[4] Microservice system benchmark, trainticket. https://github.com/FudanSELab/train-ticket/, 2018.

[5] 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.

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

[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] 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.

[9] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. Lof: identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 93–104, 2000.

[10] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *2008 eighth ieee international conference on data mining*, pages 413–422. IEEE, 2008.

[11] Linchang Fan, Jinqiang Ma, Junjing Tian, Tonghan Li, and Hao Wang. Comparative study of isolation forest and lof algorithm in anomaly detection of data mining. In *2021 International Conference on Big Data, Artificial Intelligence and Risk Management (ICBAR)*, pages 1–5. IEEE, 2021.

[12] Robert L. Thorndike. Who belongs in the family. *Psychometrika*, pages 267–276, 1953.

[13] Andrew P Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern recognition*, 30(7):1145–1159, 1997.

[14] Tatsuaki Kimura, Akio Watanabe, Tsuyoshi Toyono, and Keisuke Ishibashi. Proactive failure detection learning generation patterns of large-scale network logs. *IEICE Transactions on Communications*, 102(2):306–316, 2019.

[15] J. Zhao, N. Chen et al. Real-time incident prediction for online service systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 315–326, 2020.

[16] David MW Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. *arXiv preprint arXiv:2010.16061*, 2020.

[17] Yijun Lin and Yao-Yi Chiang. A semi-supervised approach for abnormal event prediction on large operational network time-series data. *arXiv preprint arXiv:2110.07660*, 2021.

[18] Amrit Pal and Manish Kumar. Dlme: distributed log mining using ensemble learning for fault prediction. *IEEE Systems Journal*, 13(4):3639–3650, 2019.

[19] Leonardo Mariani and Fabrizio Pastore. Automated identification of failure causes in system logs. In *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*, pages 117–126. IEEE, 2008.

[20] Xiao Yu, Pallavi Joshi, Jianwu Xu, Guoliang Jin, Hui Zhang, and Guofei Jiang. Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. *ACM SIGARCH Computer Architecture News*, 44(2):489–502, 2016.

[21] 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.

[22] 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.