Correlating Test Events With Monitoring Logs For Test Log Reduction And Anomaly Prediction

Bahareh Afshinpour, Roland Groz, Massih-Reza Amini

Correlating Test Events With Monitoring Logs For Test Log Reduction And Anomaly Prediction

1st Bahareh Afshinpour
Univ. Grenoble Alpes, CNRS, Grenoble INP
Grenoble, France
bahareh.afshinpour@univ-grenoble-alpes.fr

2nd Roland Groz
Univ. Grenoble Alpes, CNRS, Grenoble INP
Grenoble, France
Roland.Groz@univ-grenoble-alpes.fr

3rd Massih-Reza Amini
Univ. Grenoble Alpes, CNRS Grenoble INP
Grenoble, France
Massih-Reza.Amini@univ-grenoble-alpes.fr

***Abstract*—Automated fault identification in long test logs is a tough problem, mainly because of their sequential character and the impossibility of constructing training sets for zero-day faults. To reduce software testers' workload, rule-based approaches have been extensively investigated as solutions for efficiently finding and predicting the fault. Based on software system status monitoring log analysis, we propose a new learning-based technique to automate anomaly detection, correlate test events to anomalies and predict system failures. Since the meaning of fault is not established in system status monitoring-based fault detection, the suggested technique first detects periods of time when a software system status encounters aberrant situations (*Bug-Zones*). The suggested technique is then tested in a real-time system for anomaly prediction of new tests. The model may be used in two ways. It can assist testers to focus on faulty-like time intervals by reducing the number of test logs. It may also be used to forecast a *Bug-Zone* in an online system, allowing system administrators to anticipate or even prevent a system failure. An extensive study on a real-world database acquired by a telecommunication operator demonstrates that our approach achieves 71% accuracy as a Bug-Zones predictor.**

***Index Terms*—Software testing, Log analysis, Anomaly prediction.**

I. Introduction

A classical viewpoint on software testing assumes that for each given input entry (which could be a vector or sequence of inputs), the software returns an output (or a log event) record which is distinct from the other input-output (or input-log-event) pairs. Accordingly, there is a "Pass" or "Fail" verdict associated with each such input(s)-output(s) pair. A test campaign using a test suite would collect all such pairs and associated verdicts in a test log. The 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" or "test-suite reduction" [1]. 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 this type of software, in which, every action (adding items to the basket, check-out, payment) is associated with its own outputs or log events [1]. When an erroneous output is detected, software developers investigate the corresponding input to find out where, in the code, it triggers the error. Also, distinguishing the erroneous from the correct outputs/logs allows proposing supervised machine learning approaches for test/log analysis, prediction, modeling or reduction [2]. This situation is referred to as *direct logging* in this paper.

Actually, there can be some delay between a fault and its propagation to a visible output. In this case, the internal faults drive the computer system into a period of anomalous behavior which may end up in a system failure. Many complex software systems experience similar situations. For instance, a network appliance, a cellphone, or a multi-user operating system may experience a period of anomaly that ends up in a system freeze or reboot.

For a mature software system, failures may be rare. They might occur only on long software runs, either in testing conditions (e.g., with a so-called soak or endurance testing) or during system exploitation. In that case, when gathering direct logs and outputs is not feasible or impractical, a practical way to find anomalous behavior and their root cause input is *monitoring logging*. In *monitoring logging*, software testers sample the device's status or system monitoring information (called telemetry data [3], e.g: memory/CPU usage, number of processes, etc.) and then study this status information to find anomalous behavior. In this situation, finding anomalies and root causes are long and tedious tasks, if they are done manually due to the large number of log files and rare periods of anomalies [4].

In this paper, we present a chain of machine learning model creation steps, to find anomalies in status monitoring logs, link the input tests to the detected anomalies and make a system failure predictor based on the created model. The outcome is a collaborative learning approach with minimum empirical parameters that can scale up and down with the various status features and rates of sampling. This approach can be used in many applications with *monitoring logging* and operates in two steps. First, it ties some anomaly detection methods and aggregates their outputs to find anomalous periods with a high density of anomaly status, identifying them as "*Bug-Zones*". Then the system automatically extracts some tests leading to *Bug-Zones* and outside these zones at random to create a training set and a classifier is trained to associate between tests and their obtained outputs from the previous step.

This work was supported by the French National Research Agency: PHILAE project (N° ANR-18-CE25-0013).

The proposed method has two application goals. First, it filters out large test logs and extracts only the tests that are linked to anomalous behavior. This goal is favored by system developers, integrators, testers and operators as it helps them focus only on the specific periods of testing that contribute to the system failure. Second, it makes an *online predictor* to alert system administrators about 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. 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. Section II overviews the work related to our study. In section III, we present the Telecom case study. In section IV, we explain our proposed method in detail. Section V explains our implementation and empirical result of our case study. Finally, we conclude the paper in Section VI.

II. Related work

Due to the costly effort of manual log analysis, automating the testing process is highly desirable, and a great deal of research has been conducted on log analysis automation. However, in spite of their efforts, the authors of this paper failed to find related works whose proposed methods match the assumptions of this paper entirely. Therefore they can not be adopted to solve the introduced problem and be compared with the proposed method. 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 [5] 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 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 [6].

Furthermore, in the majority of approaches, the definition of fault is apparent [4] [7], while in our case, abnormal behavior 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 helpful in our case.

Among limited published research on status monitoring logs, authors of [3] 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 glassbox 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.

III. The Telecom case study

The proposed method development was motivated by a telecom case study, in which an end-user home internet box had been tested daily during six months by a large number of remote requests. The monitoring information was recorded meanwhile. The log files had two categories:

- Test logs: They record test events that arrived at the internet appliance. One log file per day contained thousands of remote requests, each of which with its timestamp. The recorded requests were from different categories of network activities such as Web surf, Digital TV, VoIP, WiFi, Software Install, P2P, Etc. For example, this is a sample of a test event recorded in the Test logs:
  ”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”.
- Monitoring Logs: The effect of the tests on internal resources like memory, CPU, processes and network traffic captured by sampling the under-the-test device. Here is a sample of 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”.

While the intervals of the test events are variable and in order of seconds, the intervals of the monitoring samples are (approximately) constant and in order of minutes, namely: 1, 5 and 10 minute(s) depending on the benches and targets, as the monitoring log collects information from several parts of the test bench. Therefore, in the period between the timestamps of two consecutive sampled statuses, hundreds of test events are recorded in the test logs. From time to time, some rare reboots occurred due to system failure. Actually, in the logs that were available, some reboots had not been caused by failure, but by a decision from the test system or test 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.

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

IV. Background

To elaborate more on the above-mentioned problems, we assume that a software system receives a chain of test events. Examples of the test events could be network packets, database queries, http requests, or API calls. Fig. 1 illustrates such a system monitoring logging conditions.

Test events are denoted by I=[I_1, …,I_N], a sequence of N events. Since the events are recorded at their arrival time, each test event I_i is a pair of *event type* which is a member of all possible test events, along with a *timestamp* that determines when the test event arrives or is executed on the system. On the observation side, the system status is recorded. That is what we call the *monitoring logging* from now on. After several test events, a *monitoring logging* observation event O_j happens that records system's status information (e.g., memory, CPU usage, etc) in an array of values or *metrics*. Each *monitoring logging* event O_j is an array of metric values and a timestamp. The monitoring log also reports some system failures noted by their timestamps. The period of status sampling is τ (Fig.1).

The goal of our method is twofold: *Bug-Zone Finder* as an indicator of the system's anomalous behavior and *Bug-Zone Predictor* as a tool to predict the imminent risk of system failure.

A. Bug-Zone *Finder*

The first part of the proposed method is the *Bug-Zone Finder*. As presented before, a *Bug-Zone* is a period of time when the software system exposes an anomalous behavior. Finding *Bug-Zones* is done in several steps:

1) Anomaly Detection: To find these periods, the first step is to deploy outlier detection functions to preprocess the monitoring data. We use a small set of 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 Windows: As shown in Fig. 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 :

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

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

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 Fig. 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}, depend 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 *Threshold and 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 Fig. 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 $BZ \rightarrow T_E$.

B. *Learning Phase*

The learning phase has three steps: Test event extraction, Model construction and 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) to investigate its root cause. 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(Fig. 1). Determining the value of τ depends on the duration that a test effect lasts on monitoring data, and it varies from one system to another. In other words, we must determine how long the test before a *Bug-Zone* should be taken into consideration. 3τ proved to exhibit the best results in our case, where sampling

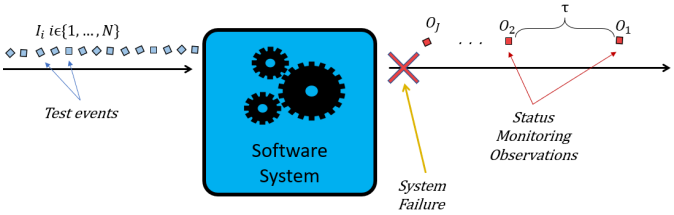


Fig. 1. A software system with input and monitoring events

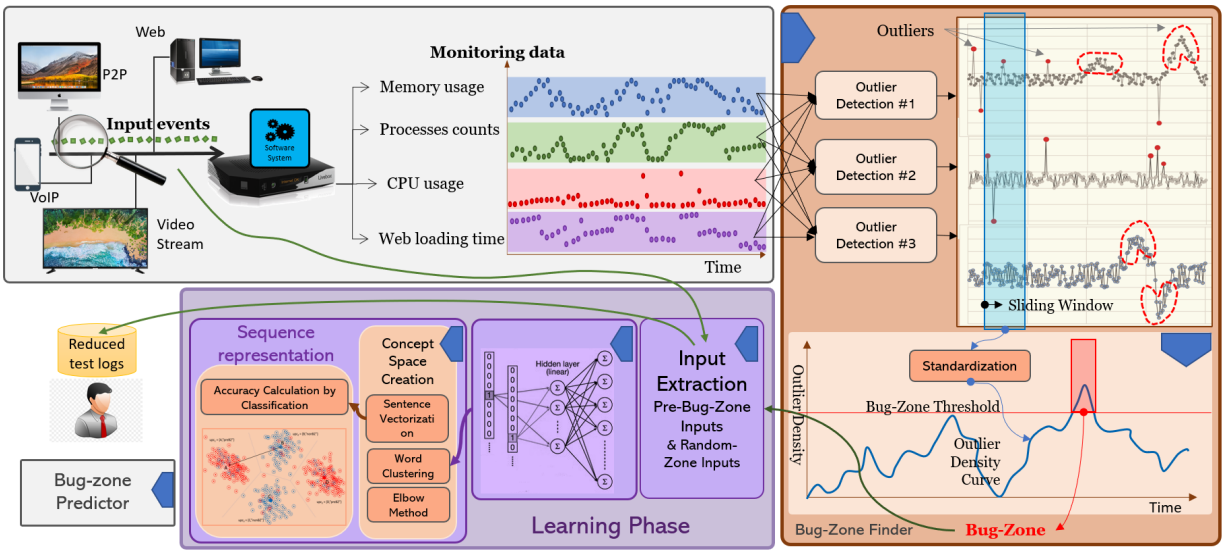


Fig. 2. An overview of the proposed method.

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 one hot coding vector. We employed a contextual sequence model proposed by [8] to learn the representation of each test event. The model then maps each type of test event into a vector. The array size is $|\phi|$, in which, ϕ is a set of all possible test event types, called *vocabulary*.

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 sequence vectors in a way that effectively reflects the semantics of the sequence. To this aim, we create 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 [9]. Then, sequences of events are mapped in the space induced by these clusters.

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:

$$PreBZ_z^{Concept} = [con_{z1}, \ldots, con_{zc}] \quad (7)$$

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. Online ML-based Bug-Zone Prediction

Online Bug-zone prediction gives an advance warning to system administrators about imminent anomalies and probable system failure. The last step to have the online predictor is to 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.

V. Evaluation on the Telecom Case Study

In this section, we evaluate the effectiveness of our approach. The aim is to answer the following research questions:

$Q1$: How accurately can our model distinguish between Pre-Bug-Zone and Random-Zone sequences?

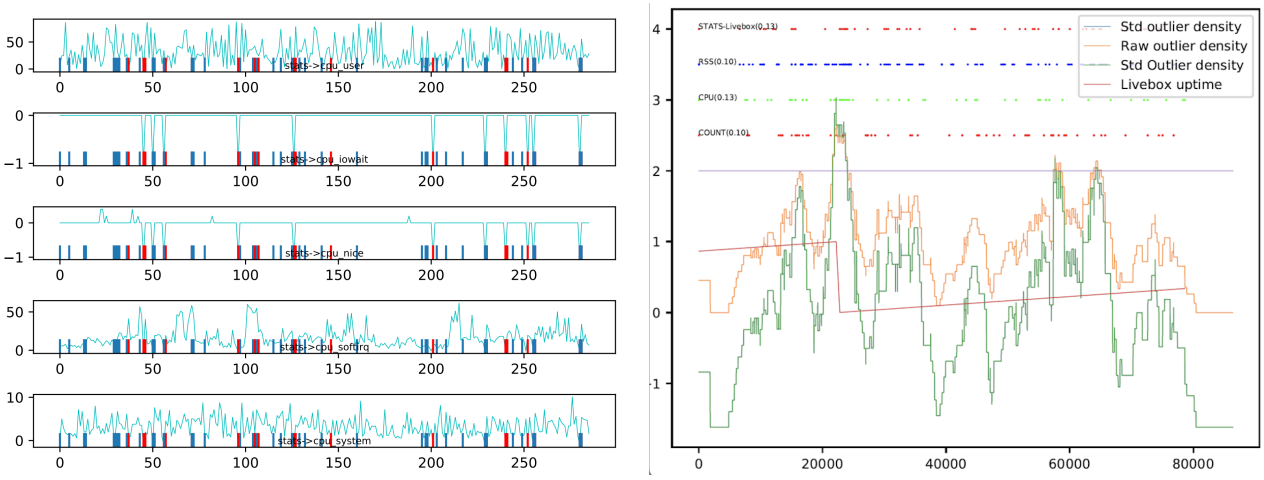
$Q2$: How effective is the proposed approach in predicting *Bug-Zones*?

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

A. Experimental Setup

We developed a python 3.x script to orchestrate and chain the proposed steps. We processed the monitoring and test data from the telecom case study by the proposed engine.

For the first step, the outlier detection engines processed the multivariate status information and determined the outlier entries. Figure 3(a) shows arrays of the "CPU" multivariate



(a) (b)

Fig. 3. (a) The first five arrays of the "CPU" status information recorded in a day, (b) Outlier density curve and detected *Bug-Zones*

status information in a light green color recorded in one day. Each array has 288 samples taken on 5-min periods during the day (288×5min = 24 hours). The CPU status information had 26 multivariate arrays, but for illustration purpose, only the first five were chosen to be plotted.

During this study, we used two different outlier detection methods, Local Outlier Factor and Isolation Forest [10], [11]. Isolation Forest is more efficient in finding global outliers and is weak in detecting local outliers. The outlier samples are depicted in Fig. 3(a) in short blue and red lines. The blue ones come from the Local Outlier Factor outlier detection and the red ones are from Isolation Forest. Noticeably, we can observe anomalies around peaks in some metrics sketches. As it is observable, in some regions, the two-outlier detection detect the same samples and in some other regions, they detect different samples. Fig. 3(a) shows how the two outlier detection methods complement each other, while there is no limit for the number of outlier detections to be used, and more outlier detections can help to accumulate all methods' detection strength.

Figure 3(b) illustrates the outlier density curve after applying the sliding windows and standardization steps. There are four rows of colorful dots scattered on top of the figures which are the outliers detected by LOF and IF outlier detection tools. Each row of dots belongs to a multivariate series of status monitoring. The fall on the uptime curves in red show a reboot in each day. The yellow curve shows the outlier density before standardization and the green shows the same after standardization. The horizontal line on 2 is the Bug-Zone threshold. Obviously around the reboot events, the threshold cuts the green curve and detects a *Bug-Zone*.

Based on our observation, 70% of the reboots were detected inside a Bug-Zone; that indicates the Bug-Zones finder is effective in predicting system failures and the relation between anomalous behavior and status monitoring is detectable by the Bug-Zone finder. The undetected reboots may have implications. They may be triggered by a hardware (or more often in that case network) failure and not be detectable by the proposed method. And we know that some of the reboots are even not bugs, they have been triggered by testers to restart sessions. Some other detected Bug-Zones were not near a reboot. Therefore, they may come from transient periods of anomalous behavior that ended without a total system failure. After getting the *Bug-Zones*, we extracted the Pre-Bug-Zone and Random-Zone sequences from the input sets. In total, we had 175 different elementary test events (that become vocabs for our NLP based approach), 589 Pre-Bug-Zone sequences and 568 Random-Zone sequences. We deployed the word embedding technique to create the NLP model. Afterward, by using K-means in combination with Elbow method [12], we created 20 concepts from the 175 vocabularies. Finally, the Pre-Bug-Zone and Random-Zone sequences are converted to their corresponding concept-space vectors, which enables us to use them for Bug-Zone prediction.

B. Q1: How accurately can our model distinguish between Pre-Bug-Zone and Random-Zone sequences?

Here, we seek to find how accurately our model can distinguish between Pre-Bug-Zone and Random-Zone sequences. A supervised classifier can determine how the Pre-Bug-Zone and Random-Zone sequences are different from one another. Since, the clusters are not linearly separated, we chose three different types of non-linear classifiers to separate them. More precisely, in this step we used concept space vectors (dim=20) of Pre-Bug-Zone and Random-Zone sequences. We employed three common classifiers in our study: Support Vector Machines (SVM), Random Forest (RF) and Multi-Layer Perceptron (MLP) from the Scikit-learn library implementations. Since the boundaries on our dataset are hypothesized to be non-linear, we chose RBF (radial basis function) as the SVM kernel function. Their accuracy to classify the Pre-Bug-Zone and Random-Zone sequences are presented in Table I. Random Forest, with 75% accuracy, has the highest rank. Fig.

TABLE I
CLASSIFICATION METHODS APPLIED ON PRE-BUG-ZONE AND RANDOM-ZONE SEQUENCES

Method	Accuracy
MLP	64%
SVM(RBF)	62%
RF	75%

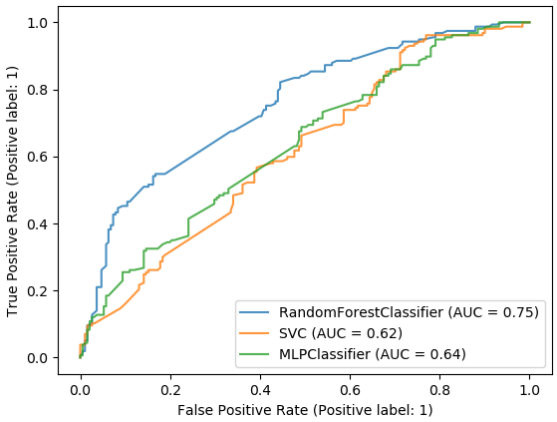


Fig. 4. The Roc curve for Random Forest, SVM and MLP classifiers

4 presents the ROCs obtained for these three classifiers. ROC curves are mostly used in binary classification to study the accuracy of a classifier [13]. This plot shows the ratio of True Positive Rate of every classifier to its False Positive Rate. 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.

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

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. We chose Random Forest for prediction since it was the most effective among the other methods in the previous subsection. Random Forest, after training, succeeded in correctly classifying 71% of the test dataset. This implies that it can be used to predict Bug-Zones based on real-time incoming test data.

Moreover, we computed common classification metrics, namely, *precision*, *recall*, and *F1-score* which are routinely used in similar work [14] [15] for analyzing accuracy. *Precision* is the percentage of correctly predicted Bug-Zones (True-Positive) over all Bug-Zone prediction (True-Positive+False-Positive): $(\frac{TP}{TP+FP})$. *Recall* is the percentage of Bug-Zones that are correctly predicted in advance among all the Bug-Zones (True-Positive+False-Negative): $(\frac{TP}{TP+FN})$. We calculated this metric because a false negative is much severer than a false positive for Bug-Zone prediction, since the cost of missing a Bug-Zone is much higher than that of investigating a false Bug-Zone. As presented in [15], F1-score $(\frac{2*TP}{2*TP+FN+FP})$ is the most used singleton metric which serves as an indicator of the model's performance. Table II shows the precision, recall and F1-score on the telecom case study by using RF classifier as classification method.

TABLE II
PERFORMANCE OF BUG-ZONE PREDICTION ON TELECOM CASE STUDY

Precision	Recall	F1-score
0.67	0.82	0.74

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

The complexity of the Bug-Zone finder phase is bounded by the outlier detection algorithms, in which, the local outlier factor algorithm has the highest complexity order of $O(J^2)$ and J is the number of monitoring samples. Likewise, the learning phase complexity is limited by the complexity of the model creation step, which is $O(N.logV)$. N is the number of words (test events) in the Pre-Bug-Zone and Random sentences, and V is the vocabulary size.

VI. 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. In this paper, we presented the Bug-Zone finder and Bug-Zone predictor, two approaches for detecting and predicting anomalous periods in a software system. First, the Bug-Zone finder, by using different anomaly detection methods, detects anomalous periods and enables 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 were evaluated on a real case study from the Orange company. The detected *Bug-Zones* cover at least 70% of the systems failures (reboots); and the Bug-Zone predictor succeeded in correctly predicting 71% of Bug-Zones in an 80-to-20 learn/test scenario. The figures are tainted by the fact that our ground truth for failures, namely system reboots, is actually overestimated, since a number of reboots (close to 30%) are indeed not linked to failures, but can be triggered by testers and test bench restarts, so we expect that our Bug-Zone finder and predictor are indeed performing even better than those figures show.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[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] 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.

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