Apprentissage collaboratif pour la réduction des log de test et la prédiction des anomalies sur les log de monitoring

Bahareh Afshinpour, Roland Groz, Massih-Reza Amini

Apprentissage collaboratif pour la réduction des log de test et la prédiction des anomalies sur les log de monitoring

Bahareh Afshinpour*[1], Roland Groz [†2], and Massih-Reza Amini [‡2]

[1,2,3]Université Grenoble Alpes

July 14, 2022

Abstract

L'identification des défaillances dans les logs de test est un problème difficile, principalement en raison de leur caractère séquentiel et de l'impossibilité de construire des ensembles d'apprentissage pour des défaillances jusqu'alors inconnues. Pour réduire la charge de travail des testeurs de logiciels, les approches basées sur des règles ont été largement étudiées comme solutions pour trouver efficacement les défaillances. Sur la base de l'analyse des log de monitoring de l'état du système logiciel, nous proposons une nouvelle technique basée sur l'apprentissage pour le diagnostic et la prédiction automatiques des anomalies. Dans un contexte où la caractérisation d'une panne n'est pas établie dans une approche de détection basée sur la surveillance de l'état du système, la technique suggérée détecte d'abord les périodes de temps où l'état d'un système logiciel rencontre des situations anormales (appelées *Bug-Zones*) sur la base de règles prédéfinies qui sont ensuite validées par un expert. La technique proposée est ensuite testée dans un système en temps réel pour la prédiction d'anomalies de tests invisibles. Dans des études de cas comparables, le modèle peut être utilisé de deux manières. Il peut aider les testeurs à se concentrer sur les intervalles de temps défectueux en réduisant le nombre de log de test . Il peut également être utilisé pour prévoir une *Bug-Zone* dans un système en ligne, permettant aux administrateurs des systèmes d'anticiper l'occurrence d'un problème potentiel.

Termes de l'index: Tests de logiciels, Analyse de log, Prédiction d'anomalies.

*bahareh.afshinpour@univ-grenoble-alpes.fr

†roland.groz@univ-grenoble-alpes.fr

‡Massih-Reza.Amini@univ-grenoble-alpes.fr

1 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 are distinct from the other input-output (or input-log-event) pairs. Accordingly, there is "Pass" or "Fail" verdict associated to 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" [AGA+20]. From this perspective, the effect of a single or a set of inputs is mapped to a limited set of outputs or log events. Therefore, many software testing improvements, especially, newly emerged machine-learning approaches hold this underlying assumption. 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 [AGA+20]. 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 and the correct outputs/logs allows proposing supervised machine learning approaches for test/log analysis, prediction, modeling or reduction [KWTI19]. This situation is referred to as *direct logging*.

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 situation. 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 reboot. In such systems, there is a time delay between a failure and the input that caused the failure. Knowing the period of anomaly and localizing its root cause input are desirable for two reasons: first, it allows system administrators to predict system failure and take measures before it happens. Second, it gives system developers a clue of the root cause input to resolve the issue in the source code of the software.

For a mature software system, failures may be rare. They might occur only on long software runs, either in testing conditions (e.g. with so called soak or endurance testing), or during exploitation of the system. In that case, when gathering direct logs and outputs is not feasible, 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 (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 [KMT20].

In this work, we present a chain of machine learning model creation steps, built around an End-to-end machine learning strategy, in order 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 various number of 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 of time with high density of anomaly status, identifying them as "*Bug-Zones*". An expert is then asked to label tests leading to *Bug-Zones* and outside of 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.

The proposed method has two application goals. First, it filters out large test logs and extracts only the tests that are linked to an anomalous behavior. This goal is in favor 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), 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 repository issued by the ANR PHILAE project.

The rest of this paper is organized as follows. Section 2 overviews the work related to our study. In section 3, we present the Telecom case study. In section 4, we explain our proposed method in detail. Section 5 explains our implementation and empirical result on our case study. Finally, we conclude the paper and discuss future work in Section 6.

2 Related work

In testing complex software systems, there are thousands of tests that run during the testing process each day. The test process produces huge test log files. Then, log file analysis methods are deployed to find software failures. Due to the costly effort of manual log analysis, automating the testing process is highly desirable. There is some work on the automation of the log file analysis to identify failures or detect root causes like [MP08]. In some approaches, the definition of fault is clear [KMT20] [AR19]. For instance, In [AR19] there are clearly identified failed and passed logs. In other cases, approaches have to study anomalous behavior since there is no obvious definition of faults. Our work falls in this second category.
In many works like [KBLP02], the experts prepared valid log profiles to have log templates. A database is created containing characteristics of a multiplicity of valid logs, which is used to find the faults. Therefore, this technique requires user-defined catalogs that describe the fault-related events.
Supervised approaches need to learn normal and abnormal patterns from labels. Therefore, they are generally useful in the case where tests can be divided into fail and pass groups, as, there is no clear definition of the faults and errors. Some studies, advocated the use of unsupervised learning techniques to automate the log analysis process. [MP08] presents an approach to automate log file analysis and root cause detection. It can identify problems by creating a finite state automaton (FSA) model from successful test sessions and com-

pare the created model against failed test session. 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 big and complex. However, FSA and similar workflow abstraction methods are shown to offer limited advantages for complex models([YJX+16]).
This paper is one of the few works that exploits system status monitoring observation for bug detection and prediction in software testing. 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 testing cases.

3 The Telecom case study

The proposed method development was motivated by a telecom case study, in which an end-user internet appliance 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 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 an test event record in the Test logs:
"timestamp":"2018-10-08T08:01:27+00:00, "metric": "loading time", "bench": "XX1", "target": "NavWeb http://fr.wikipedia.org", "status": "PASS", "domain": "Multi-services", "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"

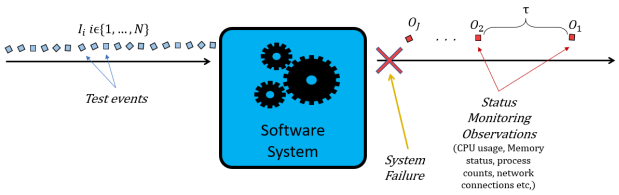


Figure 1: A software system with input and monitoring events

While the intervals of the test events are variable and in order of seconds, the intervals of the monitoring samples are 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 status, hundreds of test events are recorded in the test logs. From time to time, some rare reboots occurred due to the system failure. The manufacturer of the appliance was interested to identify 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.

4 Background

To elaborate more 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. Figure 1 illustrates such a system monitoring logging condition. 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, 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 a 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 (Figure 1).

The goal of the 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

imminent risk of system failure.

4.1 *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:

4.1.1 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 equation 1, M= (O_1,...,O_J) is the sampled multivariate monitoring data, in which, each sample O_j contains an array of metric values (as explained in previous subsection). 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.

4.1.2 Sliding Windows

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 laying 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 in Figure 3 . A_{ac}[t] is the number of all "1"s in a window by 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.

4.1.3 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})

4.1.4 *Bug-Zone* Threshold and Extraction

After standardization, by fixing the threshold value between 0 and 1, *Bug-Zones* are detectable from *ODC*. Bug-Zones are the moments when the outlier density curve rises above the threshold horizontal 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 $BZ \rightarrow T_E$. An expert is then asked to label tests in bug-zones and others picked randomly outside of these zones to create a training set.

4.2 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.

4.2.1 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 for its root cause. But we will need also 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*. For simplicity, 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

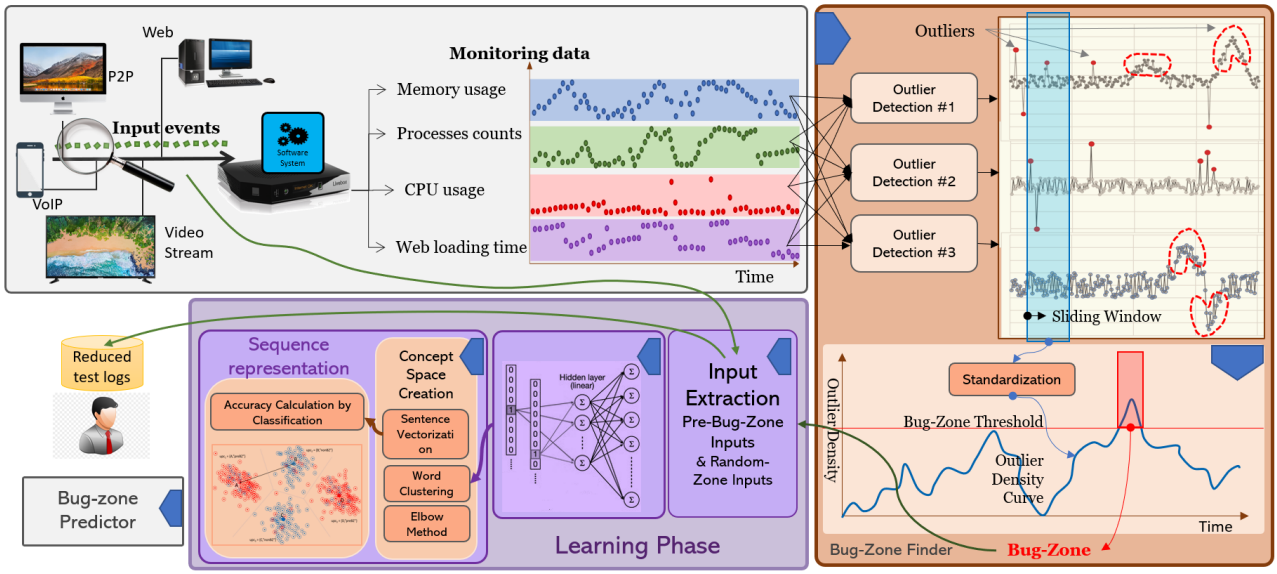


Figure 2: An overview of the proposed method.

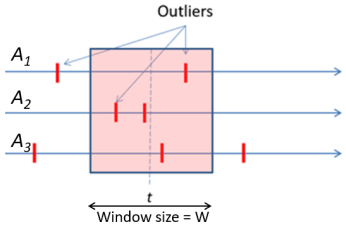


Figure 3: A sliding windows over anomaly detection arrays counts all True values

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.

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 equations 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.

4.2.2 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 [MCCD13] 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*.

4.2.3 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:

$$Rand_z^V = [I_{z1}^V, \ldots, I_{zP}^V],$$

$$PreBZ_z^V = [I_{z1}^V, \ldots, I_{zR}^V].$$

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. 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 [PKAG10]. Then, sequences of events are mapped in the space induced by these clusters. Each characteristic of the vector corresponds to the proportion of events of that cluster present in the sequence. We employed Gaussian mixture models where the number of clusters is determined by penalizing model complexity using the Akaike information criteria (AIC) [Aka74]. Other clustering methods can be used, such as spectral clustering with minimal description length (MDL).

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:

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

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 $Rand_z^{Concept}$.

4.3 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 distinguish them from the normal (Random-Zone) sequence.

5 Implementation and Results on the Telecom Case Study

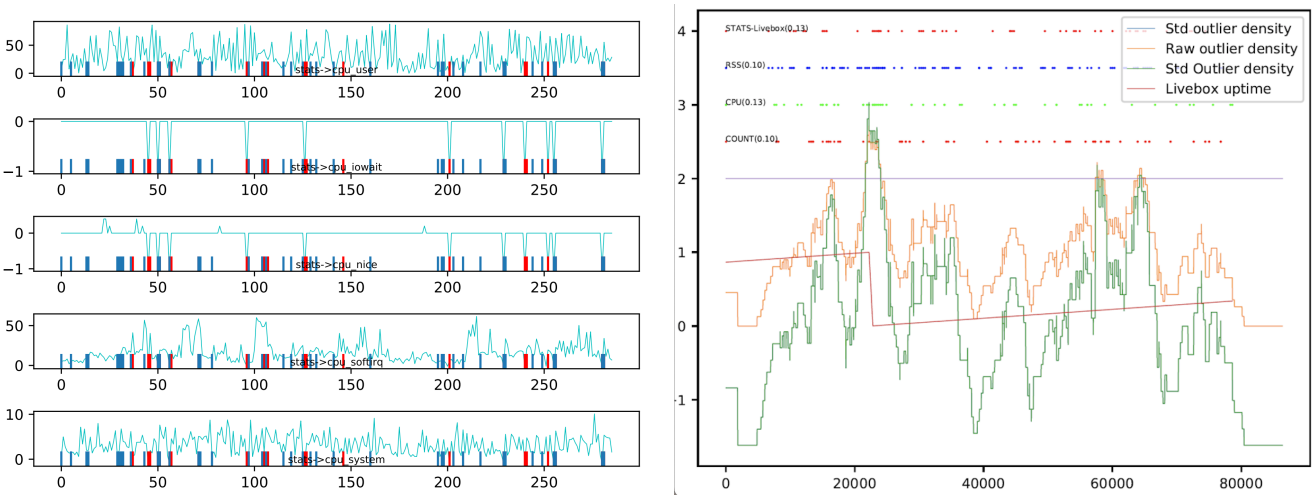
We developed a python 3.x script to orchestrate and chain the proposed steps. We processed the monitoring and test data from the telecom case study by the proposed engine. The results of each step will be presented in the following subsections:

- *Outlier Detection*

For the first step, the outlier detection engines processed the multivariate status information and determined the outlier entries. Figure 4(a) shows arrays of the "CPU" multivariate status information in a light green colour 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 [BKNS00, LTZ08]. The outlier samples are depicted in Figure 4(a) in short blue and red lines. The blue ones come from the Local Outlier Factor outlier detection and the red ones from Isolation Forest. Noticeably, we can observe anomalies around peaks in some metrics sketches. As is observable, at some regions, the two-outlier detection detect the same samples and at some other regions they detect different samples. The Figure 4(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.

- *Bug-Zone Finder*

Figure 4(b) illustrates the outlier density curve after applying the sliding windows and standardization steps. There are four rows of colourful 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 of 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 failure and not be detectable by the proposed method. Or some of reboots are even not bugs, they have been triggered by testers to restart sessions. Some other detected Bug-Zones where not near a reboot. Therefore, they may come from transient periods of anomalous behaviour ended



(a) (b)

Figure 4: (a) The first five arrays of the "CPU" status information recorded in a day, (b) Outlier density curve and detected *Bug-Zones*

without a system failure.

- *Model Creation and Sequence Representation*

After having the *Bug-Zones*, we extracted the Pre-Bug-Zone and Random-Zone sequences from the inputs sets. In total, we had 175 different tests (vocabs), 589 Pre-Bug-Zone sequences and 568 Random-Zone sequences. We deployed word embedding technique to create the NLP model. Afterwards, by using K-means in combination with Elbow method, we created 20 concepts from the 175 vocabularies. Finally, the Pre-Bug-Zone and Random-Zone sequences converted to their corresponding concept-space vectors that enables us to use them for Bug-Zone prediction.

- *Visualization of the vectors*

A visualization of the sequence vectors which were created in the previous step, helps us to have an insight on how they are scattered and what type of classifier is needed. To visualize high dimensional vector, we used t-SNE [VdMH08], a dimensional reduction tool, to map the vectors in 2D plane. The plotted image of the vectors are depicted in Figure 5. Each red dot represents a Pre-Bug-Zone test sequences, and likewise, each blue dot is a Random-Zone test sequences. Obviously, there are clusters of sequences with distinction, in some of which, the majority of dots are either red or blue. The are some mixed clusters with nearly equal number of red and blue dots. Despite their mixed 2D image, they may not be mixed in higher dimensions. This can be evaluated by a classifier.

- *Preliminary Results and Efficiency of the Model*

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 in Figure 5, 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

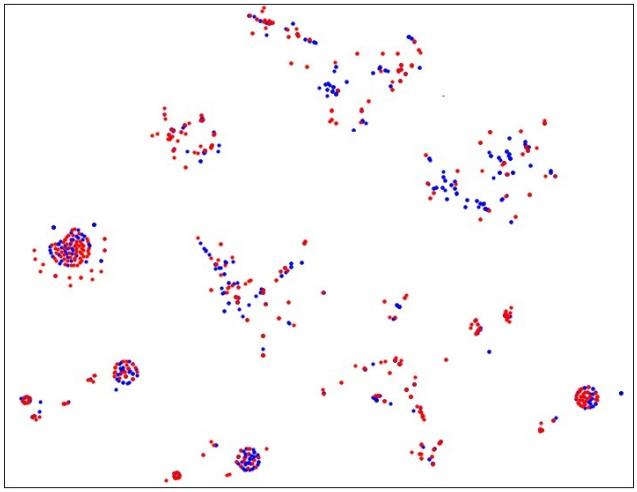


Figure 5: Concept-space vectors after dimension reduction by t-SNE - red dots represent a Pre-Bug-Zone and blue dots represent Random-Zone sequences

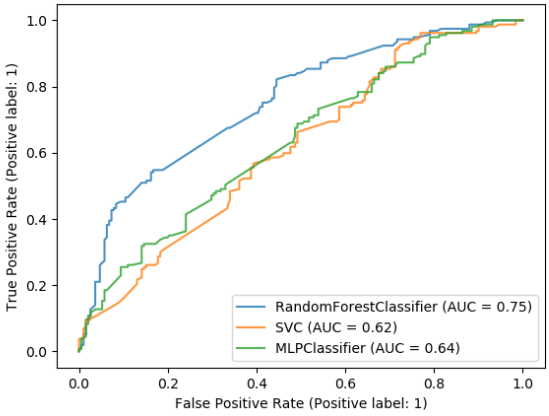


Figure 6: The Roc curve for Random Forest, SVM and MLP classifiers on concept space vectors

sequences. We employed three common classifiers in our study: Support Vector Machines (SVM), Random Forest (RF) and Multi-Layer Preceprton (MLP) from the Scikit-learn library implementations. All these three approaches belong to the category of supervised algorithms. Since the boundaries on our dataset are hypothesized to be non-linear, we chose RBF (radial basis function) as the SVM kernel function. To assess the classification accuracy, we applied a 10-fold validation approach. Their accuracy to classify the Pre-Bug-Zone and Random-Zone sequences is presented in Table 1. Random Forest, with 75% accuracy, has the highest rank. Figure 6 presents the ROCs obtained for these three classifiers. ROC curves are mostly used in binary classification to study the accuracy of a classifier [Bra97]. This plot shows the True Positive Rate of every classifier as a function of the False Positive Rate of the same classifier. The results show that Random Forest classifier outperforms the other classifiers. 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.

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

Table 1: Classification methods applied on Pre-Bug-Zone and Random-Zone sequences

- Bug-Zone-Prediction and results

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 to correctly classify 71% of the test dataset. This implies that it can be used to predict Bug-Zones based on a real-time incoming test data.

A model can estimate the probability of data belonging to each class label. We used cross-entropy to calculate the difference between the two probability distributions in our classification. The Average cross Entropy is 0.5 nats. It demonstrates that the model is making a reasonable generalisation about the data as the validation scores are not significantly high.

6 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 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 two different anomaly detection methods, allowed us to detect anomalous periods and extract *Bug-Zones*. This 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 end-to-end ML technique to create a conceptual vector from the semantics of the test sequences, our online predictive model is able to identify sequences of tests that lead to 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 70% of the systems failures (reboots) and the Bug-Zone predictor succeeded to correctly predict 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 testbench restarts, so we expect that our Bug-Zone finder and predictor are indeed performing even better than those figures show.

7 Acknowledgment

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

References

[AGA+20] 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.

[Aka74] H Akaike. *A new look at the statistical model identification*, volume 19. 1974.

[AR19] 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.

[BKNS00] 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.

[Bra97] 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.

[BRST17] Christophe Bertero, Matthieu Roy, Carla Sauvanaud, and Gilles Trédan. Experience report: Log mining using natural language processing and application to anomaly detection. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 351–360. IEEE, 2017.

[GU16] Markus Goldstein and Seiichi Uchida. A comparative evaluation of unsupervised anomaly detection algorithms for multivariate data. *PloS one*, 11(4):e0152173, 2016.

[KBLP02] Anurudha Kulatunge, Kalyan Basu, Hee C Lee, and Meenakshi Prakash. Network fault prediction and proactive maintenance system, March 5 2002. US Patent 6,353,902.

[KMT20] 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.

[KWTI19] 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.

[LTZ08] 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.

[MCCD13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[MP08] 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.

[MSC+13] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[PK19] Amrit Pal and Manish Kumar. Dlme: distributed log mining using ensemble learning for fault prediction. *IEEE Systems Journal*, 13(4):3639–3650, 2019.

[PKAG10] 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.

[Tho53] Robert L. Thorndike. Who belongs in the family. *Psychometrika*, pages 267–276, 1953.

[VdMH08] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[XHF+09] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117–132, 2009.

[YJX+16] 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.

[YLL+20] En-Hau Yeh, Phone Lin, Xin-Xue Lin, Jeu-Yih Jeng, and Yuguang Fang. System error prediction for business support systems in telecommunications networks. *IEEE Transactions on Parallel and Distributed Systems*, 31(11):2723–2733, 2020.