Mining Software Logs with Machine Learning Techniques

Bahareh Afshinpour

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : Laboratoire d'Informatique de Grenoble

Exploitation de journaux logiciels avec des techniques d'apprentissage automatique

Mining Software Logs with Machine Learning Techniques

Présentée par :

Bahareh AFSHINPOUR

Direction de thèse :

Roland GROZ PROFESSEUR DES UNIVERSITES, GRENOBLE INP	Directeur de thèse
Massih-Reza AMINI PROFESSEUR DES UNIVERSITES, Université Grenoble Alpes	Co-directeur de thèse

Rapporteurs :

NEIL WALKINSHAW
ASSOCIATE PROFESSOR, UNIVERSITY OF SHEFFIELD
FRANZ WOTAWA
PROFESSEUR, TECHNISCHE UNIVERSITÄT GRAZ

Thèse soutenue publiquement le **29 septembre 2023**, devant le jury composé de :

ROLAND GROZ PROFESSEUR DES UNIVERSITES, GRENOBLE INP	Directeur de thèse
NEIL WALKINSHAW ASSOCIATE PROFESSOR, UNIVERSITY OF SHEFFIELD	Rapporteur
FRANZ WOTAWA PROFESSEUR, TECHNISCHE UNIVERSITÄT GRAZ	Rapporteur
OUM-EL-KHEIR AKTOUF PROFESSEURE DES UNIVERSITES, GRENOBLE INP	Présidente
ARNAUD GOTLIEB Chief Research Scientist / Research Professor, SIMULA RESEARCH LABORATORY	Examinateur
MASSIH-REZA AMINI PROFESSEUR DES UNIVERSITES, UNIVERSITE GRENOBLE ALPES	Co-directeur de thèse

Acknowledgement

I would like to take this opportunity to express my gratitude to a number of people who supported me on this research journey. First and foremost, I was very lucky to have Roland Groz as my supervisor, as he helped me in many aspects of my studies and my life. His feedback on my work from various perspectives was extremely useful for me. Also, I would like to express my gratitude to my co-supervisor Massih-Rezza Amini for his encouragement and inspiration. I am thankful for their devotion, assistance, and advice. I have learned a lot from them, and in many instances helped me regain confidence in myself. It was an honor to be part of their team.
I want to express my appreciation to Yves Ledru, who served as the Vasco director and provided me with insightful advice and technical guidance. I am very thankful for his discussions and feedback. I have enjoyed collaborating with numerous colleagues and friends from various teams, mainly Vasco and Aptikal. In particular, special thanks and appreciation go to Lydie Bousquet and German Vega for their valuable time and assistance.
I am grateful to the members of my defense committee, especially Neil Walkinshaw and Franz Wotawa for dedicating time and effort to reviewing my dissertation.
Apart from the academic aspect of the process of learning, there is my family, without whom I wouldn't have reached this stage. Therefore, I wish to express my appreciation to my parents for their unwavering support throughout my existence. They're always believing in me. Thank you for the unconditional love and support I continue to receive from you.
Lastly, but by no means least, I would like to thank my husband for the numerous ways in which he supported me, from listening to me to encouraging me to do my best in my studies.

Bahareh Afshinpour
La Tronche, September 20, 2023

Dedication

I dedicate my thesis work to my husband,

Ehsan

and my wonderful sons,

Hooman and **Homayoun**

who have supported me in both pleasant and difficult times since the beginning of my studies.

Abstract

Today, software is used in a wider variety of areas than ever before. Testing software is one of the techniques utilized during the verification and validation process. Researchers and the business sector have attempted to automate software testing in the past few decades, as the majority of testing activities are difficult and costly.
In recent decades, software logs have become indispensable to the reliability assurance mechanism of many software systems, as they are typically the only data that records software runtime events. They are a valuable information source that can be leveraged for various diagnostic purposes. Throughout the testing procedure, testers can extract vital information from logs.
Regression tests are required to be executed after each iteration of software development, which can be costly in terms of time and resources. Additionally, the volume of logs has rapidly increased as the applications of modern software have grown. The regression testing process needs to be automated in order to mitigate the cost associated with log analysis and reduce the workload of software testers.
Log mining employs statistics, data mining, and machine learning techniques to automatically investigate and analyze a large volume of log data in order to discover meaningful patterns and reveal trends. Advanced implementation strategies for automated log mining are in high demand. Log mining tasks related to automated software testing are one of the contributions of this dissertation. We introduce some major log mining tasks for reliability engineering, including anomaly detection, failure prediction, and root-cause detection, etc. The research is completed through a number of case studies and experiments, which ultimately leads to the development of a set of tools that work together to help automate log mining. The results given in this dissertation demonstrate how software testing can be enhanced by employing log mining using machine learning. This dissertation introduces four important log mining problems, including root-cause detection, online failure prediction, log minimization, and user behavior clustering. Based on software system log analysis, we propose a new learning-based technique to automate log mining tasks. A part of the effort in this work focuses on developing unsupervised log mining methods in order to reduce human interaction and extract hidden features that are hidden from direct human observation. To this end, we tried to adopt learning techniques (e.g., NLP) that are capable of extracting the semantics of the logs and, therefore, learning the relations among the events. This evolved into a general unsupervised log mining methodology capable of clustering output events based on their conceptual relations with other events, detecting anomalous behavior, predicting them in online software, and finally finding their root cause among the input events. The achievements of the thesis help system administrators predict the possibility of imminent failures and also help software developers detect bugs and their root causes among input and output log records.
Throughout this dissertation, "real-world" applications are discussed, and we believe that our work could serve as the foundation for future research and deployment of automated log mining, as well as provide important recommendations in this area.

Keywords Automated software testing, Log analysis, Machine learning, Log mining tasks

French Résumé

Aujourd'hui, les logiciels sont utilisés dans une plus grande variété de domaines que jamais auparavant. Le test logiciel est l'une des techniques utilisées au cours du processus de vérification et de validation. Les chercheurs et le secteur des entreprises ont tenté d'automatiser les tests de logiciels au cours des dernières décennies, puisque la majorité des activités de test sont difficiles et coûteuses. Au cours des dernières décennies, les journaux de logiciels sont devenus indispensables au mécanisme d'assurance de la fiabilité de nombreux systèmes logiciels, car ce sont généralement les seules données qui enregistrent les événements d'exécution des logiciels. Tout au long de la procédure de test, les testeurs peuvent extraire des information vitales des journaux. Les tests de régression doivent être exécutés après chaque itération de développement logiciel, ce qui peut être coûteux en temps et en ressources. De plus, le volume de journaux a rapidement augmenté à mesure que les applications des logiciels modernes se sont développées. Le processus de test de régression doit être automatisé afin d'atténuer les coûts associés à l'analyse des journaux et de réduire la charge de travail des testeurs de logiciels. L'exploration de journaux utilise des statistiques, l'exploration de données et des techniques d'apprentissage automatique pour analyser automatiquement un grand volume de données de journaux afin de découvrir des modèles significatifs et des tendances révélatrices. Les stratégies de mise en œuvre avancées pour l'extraction automatisée de journaux sont très demandées. Les tâches d'extraction de journaux liées aux tests automatisés de logiciels sont l'une des contributions de cette thèse. Nous introduisons certaines tâches majeures d'extraction de journaux pour l'ingénierie de la fiabilité. La recherche est complétée par un certain nombre d'études de cas et d'expériences, ce qui conduit finalement au développement d'un ensemble d'outils qui travailler ensemble pour aider à automatiser l'extraction de qui travaillent ensemble journaux. Les résultats donnés dans cette thèse démontrent comment les tests de logiciels peuvent être améliorés en utilisant l'extraction de journaux à l'aide de l'apprentissage automatique. Cette thèse présente quatre problèmes importants d'extraction de journaux, notamment la détection des causes, la prédiction des défaillances en ligne, la minimisation des journaux de test et le regroupement du comportement des utilisateurs. Basée sur l'analyse des journaux du système logiciel, nous proposons une nouvelle technique basée sur l'apprentissage pour automatiser les tâches d'extraction de journaux. Une partie de l'effort dans ce travail se concentre sur le développement de méthodes non supervisées d'extraction de zones afin de réduire l'interaction humaine et également d'extraire des caractéristiques qui sont cachées à l'observation humaine directe. À cette fin, nous avons essayé d'adopter des techniques d'apprentissage (par exemple, NLP) capables d'extraire la sémantique des journaux et, par conséquent, d'apprendre les relations entre les événements. Cela a évolué vers une méthodologie générale d'exploration de journaux non supervisée capable de regrouper les événements de sortie en fonction de leurs relations conceptuelles avec d'autres événements, de détecter un comportement anormal, de les prédire dans un logiciel en ligne et enfin de trouver leur cause première parmi les événements d'entrée. Cela aide les administrateurs système à prévoir la possibilité de pannes imminentes et aide aussi les développeurs de logiciels à détecter les bogues et leurs causes profondes dans les enregistrements des journaux d'entrée et de sortie. Tout au long de cette thèse, des applications logicielles issues d'un projet partenarial sont discutées et nous pensons que notre travail pourrait servir de base à la recherche et au déploiement futurs de l'extraction automatisée de journaux, ainsi qu'à fournir des recommandations importantes dans ce domaine.

mots-clefs Tests logiciels automatisés, analyse de journaux, apprentissage automatique, tâches d'exploration de journaux

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

This chapter will introduce the topic of this dissertation, as well as the motives for its development, aims, and contribution to the area. The following chapters' structure is presented at the end.

1.1 Motivation

Software testing is one of the most important phases of the software development lifecycle. It is used to detect software flaws and ensure that software is delivered in a high-quality condition. Any changes to a software component may affect one or more other components, requiring the re-execution of previously generated test cases in addition to the newly generated ones [1]. Regression testing is a software testing technique that verifies that an application continues to perform as expected after any code modifications, updates, or improvements. It should be performed after each iteration of software development, which can be expensive in terms of time and resources.

Testing information systems has become a serious bottleneck for many large corporations and small and medium-sized enterprises. Aside from the ever-increasing complexity of such systems, their unavoidable quality assurance requirements have resulted in drastically increased verification and validation expenses. However, a fine-grained review of existing testing techniques suggests that not all artifacts are being used to mitigate this cost increase. Validation engineers frequently disregard test and operational execution traces. This is hardly surprising given that these traces are nearly impossible to classify and examine by hand. Furthermore, with model-based testing methodologies, test models used to produce test cases are difficult to maintain and evolve. As a result, they are frequently abandoned and replaced by test scripts written from scratch, increasing validation costs.

It is thus critical to provide more intelligent and cognitive automated test processes in order to regulate the complexity growth of software verification activities and help to break the testing bottleneck. This will allow test engineers to concentrate on developing higher-value tests for specific scenarios.

Software logs are a precious source of information that can be exploited for different diagnostic purposes. Logging appeared as an important early component of computing systems, because it helps application developers and users figure out what occurred during program execution. Traditionally, log mining has been as simple as a search for *"error"*, *"fault"* or *"exception"* keywords. Over the last two decades, log mining has turned toward rules-based comparison against a manually created rule set to discern between normal and abnormal behavior. This trend, however, is error-prone, labor-intensive, and lacks scalability due to the ever increasing volume and complexity of software logs. By the advances of machine learning in several fields over the previous decade, log mining has been extensively explored to provide new dimensions to log analysis.

The PHILAE project[1] inspired and defined this thesis. The PHILAE project has set broad and diversified goals for software testing, which will be discussed in depth in the next sections. The PHILAE project includes various case studies, which are typically software traces supplied by the project's industrial partners. As a PHILAE project partner, the computer science laboratory in Grenoble (LIG)[2] has concentrated on specific case studies and so addressed a subset of the PHILAE objectives. As a result, this thesis at LIG was to address these aims. The goals are broadly stated as test selection and creation from usage records, as well as defect reporting and anomaly identification from logs.

[1] PHILAE was supported by the French National Research Agency: PHILAE project (N° ANR-18-CE25-0013).
[2] https://www.liglab.fr/en

1.2 Problem Statement and Research Objectives

As discussed in the previous section, we made a choice of two PHILAE case studies during the preliminary steps of this thesis. Aligned with PHILAE's objectives and the requirements of the case studies, we created a set of objectives to fulfill. We address these objectives as **log mining tasks**, considering that the case studies were basically a set of logs and also knowing that all the objectives fall into the *log mining* (or log analysis) field of research in the related literature. Here we introduce them briefly and a complete discussion will be postponed to the coming sections:

In this thesis, with a focus on the PHILAE's objectives and the requirements of the chosen case studies, we pursue the following log mining tasks using machine learning approaches:

- **Log File and Test Suite Minimization**:

  The most significant source of information for bug analysis and failure diagnosis is software logs. Following a bug or system crash report, software developers must examine log files to determine the possible cause of the incident. For large software systems, log files may include a massive number of events resulting from intertwined traces of activity made by various users. Such a system could be a cloud computing platform. Reduce uninteresting and unrelated log events is a typical approach for accelerating log analysis. As a result, determining whether one or a specific collection of events in the log file is linked to the bug occurrence is an important aspect of log minimization. Rerunning the entire chain of events is obviously the worst-case situation and is not an option.

  We can see parallels between log reduction and test suite minimization, when we need to minimize a group of tests. A test suite is a container that contains a collection of tests that assist testers in executing and reporting test execution status. A test suite is a set of test scenarios that address numerous capabilities that are crucial to the product in software testing, particularly regression testing. Test suites are often built from previously completed functional tests, unit tests, integration tests, and other test cases. A test suite is prepared after each modification to the software to ensure its functionality.

  Generally, the duration of regression testing is determined by the size of the test suites. As the size of regression testing increases, its execution becomes increasingly computationally intensive. Regression testing necessitates the execution of a large program on a large number of test cases, which can be costly in terms of both human and machine time. Test Suite Minimization or Test Suite Reduction (TSR) provides more efficient and simpler test suite maintenance, which in turn reduces the cost of the software testing phase, although in terms of the ability to detect faults. These methods work by identifying and removing obsolete or redundant test cases. Therefore, a number of different methods have been studied to deal with test suits, such as minimization and selection.

  Regression testing consumes a significant amount of time in many software projects, slowing development. By reducing redundant test cases, test suite minimization can be used to reduce the time it takes for each test run. However, in practice, test suite minimization is rarely used. We discovered two primary reasons for this. The first is that it is a difficult task to complete, especially with sophisticated builds. The second point is that deleting test cases always has the potential of lowering the test suite's efficacy. Because of the nature of test suite minimization, tests are typically removed permanently, which is a risk that must be accepted in comparison to test case selection or prioritizing. In this regard, log files are records of software events that occurred in the past, and their minimization is beneficial for diagnosis and root cause discovery, whereas test suites are records of software events that

will occur in the future, and their minimization is beneficial for saving execution time. Running the entire record of occurrences is the worst case situation in both scenarios and is not wanted. As a result, the minimization and reduction are performed on runs of software events, and the issue statements are similar in both circumstances.

- **Log Anomaly Detection**: Logs are generated by software systems to record events and the present state of the system. Because of its simplicity and effectiveness, logging has become widely used in practice. Logs are an important and valuable source of information for developers and operators, who can review recorded logs to understand the system state when troubleshooting by recognizing system abnormalities and locating the root causes. In the age of cloud computing, even mundane operations such as billing can be based on logs that record the customer's use of the service.

  Modern systems are scaling up and migrating to cloud-based distributed processing. These large-scale systems provide online services such as search engines, social networks, and e-commerce, as well as computation-intensive applications such as weather forecasting. Many of these systems are designed to run continuously and serve millions of users worldwide. Any decline in quality or even outage of such services is extremely costly, hence rapid detection of any changes and the capacity to quickly determine the root cause of the problem are critical. However, these systems generate massive amounts of log data at rates of tens of terabytes each hour. Even with search and filtering technologies, such volume is difficult, if not impossible, to manually examine.

  In response to the expanding volume of logs, automated log analysis and anomaly detection have emerged as key research topics in recent years. Automation promises online monitoring and rapid anomaly warning, allowing developers and operators to focus solely on problem solving. However, that is still in the future. With the volume of records generated by these systems, even a low ratio of false positive alerts might overload operators. And log-based anomaly identification has proven to be difficult. The majority of the critical information is concealed in log messages, which are typically unstructured or semi-structured text strings that are difficult for algorithms to comprehend. Also, log-based anomaly detection must frequently deal with a continually changing environment as a result of regular system updates.

  The task of detecting system anomalous patterns that do not match predicted behaviors using log data is known as anomaly detection. Anomalies in software systems are frequently indicative of an error, defect, or failure. Currently, logs, which preserve comprehensive information about computational events generated by computer systems, play a crucial role in anomaly detection. Traditionally, developers (or administrators) manually inspect logs using keyword search and matching rules. The increasing scale and complexity of modern systems, however, cause the volume of logs to increase, making manual inspection impossible. Therefore, numerous anomaly detection methods based on automated log analysis are proposed to reduce manual effort.

- **Failure Prediction**: Because of the rapid development of software technology, the quality of industrial applications has substantially improved. With the support of error-free software, this ever-expanding technology promotes the organization's growth. Software failure prediction [2] is required for developers to increase software quality. A single flaw in software can cause a big problem, resulting in the loss of a company's life. Although manual software testing is used in the industry, it is quite sophisticated and requires people to execute software testing. However, numerous automated prediction techniques have recently accomplished

algorithms such as Software failure prediction [3]. They are important in automatically picking relevant prediction models and other required ways to predict the number of flaws in the software module. However, because the number of parameters in each data set varies, there should be an appropriate model for each data collection [4].

Anomaly detection seeks to identify abnormal status or unexpected behavior patterns that may or may not result in failures. Failure prediction, on the other hand, seeks to create early warnings to avert server failures, which frequently result in unrecoverable states. By analyzing past system logs and identifying the relationship between the data and the failures, numerous machine-learning methods for predicting task or job failure have been proposed.

Failure prediction can be seen from another point of view. In many safety-critical software applications, predicting failures before their arrival provides more slack time for safer system shutdown. Until recently, safety-critical systems in air traffic control, commercial aircraft, and nuclear power were composed of a monolithic (potentially proprietary) system offered by a single vendor. As a result, such systems incurred substantial development and maintenance costs. To cut costs, systems have been disassembled into a collection of applications/services (often produced by separate vendors) that interact via a set of well-defined interfaces.

Applications must meet severe Quality of Service (QoS) availability standards in order to ensure the overall high availability of the safety-critical system. To accomplish this goal, applications must distribute and replicate data (for example, flight paths in an Air Traffic Control system) across multiple nodes connected by a WAN or a LAN. Because of the nature of such systems, replicas of an application must be strictly consistent in order to maintain the same state throughout time, giving a client the appearance that its request is being processed instantly [5].

Failures are a fact of life in such large distributed systems, thus they must be managed carefully to ensure system survival. Extensive testing during the design phase of an application cannot prevent the development of faults that can have disastrous effects on the whole system's operation during the operational phase. In the presence of replica failures, keeping a set of replicas completely consistent boils down to solving the consensus problem. Thus, if the distributed system has good coverage of synchrony assumptions (i.e., network and computing nodes are functioning properly), there are a number of fault tolerance mechanisms that can be used to overcome failure and keep replicas consistent (e.g., failure detection via heart-beating). If a fault occurs during a period when the synchrony assumption is not covered, replicas may exhibit anomalous behavior due to the well-known Fischer-Lynch-Paterson (FLP) impossibility result, which states that distributed consensus cannot be reached in an asynchronous system even in the presence of a single faulty process. These actions could have a knock-on effect on other applications, resulting in a system failure, such as an irregular system shutdown. In this instance, it may take a long time for the system to restore normal operation, drastically lowering system availability. The only approach to avoid such aberrant system breakdowns is to foresee them by detecting anomalous activities. Predictions that are accurate and timely can help to lessen the impact of failures by initiating suitable recovery activities before the failure happens. Such procedures can help to alleviate the loss of availability by shortening the time required to restore normal system behavior.

- **Root-Cause Event Detection (Failure Diagnosis)**: The term "Root Cause Analysis" (RCA) refers to a collection of methods and procedures used to identify the problems that lead to

an observed failure. Analyzing log files to find anomalies from normal or expected behavior is a frequent RCA strategy. Failure diagnosis, as opposed to anomaly detection and failure prediction, which are typically characterized as classification tasks, seeks to uncover the underlying reasons for a failure that has affected end users. It is frequently associated with root cause analysis. Specifically, while anomaly detection and failure prediction can determine whether an issue exists or will occur, there is a significant time lag between detecting and removing a problem or failure. RCA performance is limited by the size of the logs considered. When large software systems become involved, the issue is more obvious.

Here, it is important to distinguish between root-cause detection in incoming *events* and root-cause detection in software *source code*. In this study, the objective of RCA is to identify the relationship between incoming events (such as inputs, network requests, new connections, new user logins, etc.) and any anomalous software behavior. After conducting this research, for instance, we might discover that the failure happens following a certain remote user login attempt. This is the first stage of software testing, which aims to identify an action (or series of actions) that results in abnormal behavior or failure of the software. A second stage of source code analysis, carried out by software developers, may then follow; however, the focus of this thesis is on the first stage. We chose Root-Cause *Event* Detection as the name of this subchapter for that reason.

- **User Behavior clustering**: Recently, most software systems collect activity logs that can be organized in user traces, which represent the behavior of users on those software systems. User behavior insights could be very helpful for tasks like task automation [6] which tries to identify and automate repetitive user actions, and usability engineering [7], which analyzes how software is used and may be improved. These details can be discovered by reviewing data on user interactions with software application user interfaces (UI). UI logs can be analyzed using methods from the field of process mining, such as applying process discovery to create a process model as a visual representation of the observed user behavior [8]. Yet, it is difficult to directly acquire specific insights due to the complexity of UI logs, which is reflected in their size and behavioral variance. Clustering categorizes user traces from the System Under Test (SUT) into groups based on similar behavior. We chose UBM as one of the log mining goals of this research because we found informative clusters of users during the analysis of log files in our case studies. Since ML-based UBM has received a great deal of attention [9] recently, we decided to investigate this task in our log analysis.

1.3 PHILAE Project

The PHILAE project[3] began in 2018 with six partners from various countries' research laboratories and software companies, namely:

- FEM – Université Bourgogne Franche-Comté / Institute FEMTO-ST – UMR 6174
- LIG – Université Grenoble Alpes / Laboratoire d'Informatique de Grenoble – UMR 5217
- OLS – Orange Labs Services
- SMA – Smartesting Solutions & Services
- SRL – Simula Research Laboratory – CERTUS center - Norway

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

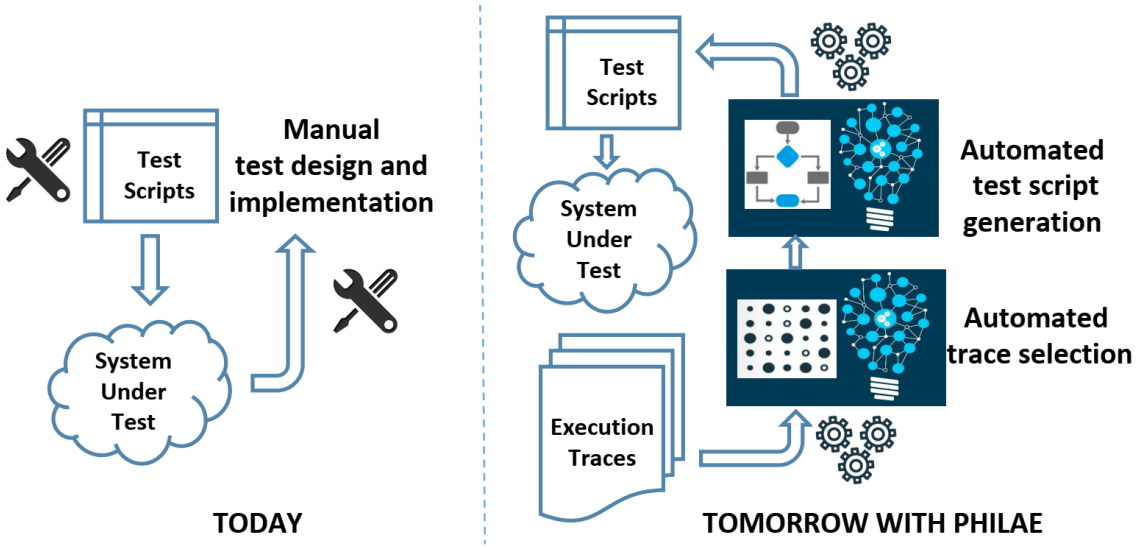


Figure 1.1: PHILAE project vision

- Flexio Logo FLE – Flexio, experimentation partner, Besançon France.
- The University Of Queensland, Australia

PHILAE's goal was to alleviate the testing bottleneck by performing trace analysis and triage with machine learning techniques, combining this analysis with model inference and automated test generation. Thus, PHILAE aims at leveraging data available from development (and validation) and usage of software systems to automatically adapt and improve regression tests. We intend to change the state of practice in automated testing of information systems by carefully transitioning from Model-Based Testing to Cognitive Test Automation by demonstrating this key-enabling approach on five industrial case studies, using an industry-strength tool-chain developed in the project Figure 1.1.

Figure 1.2 depicts the key **project process** in four iterative and incremental steps:

1. Execution traces from the running system, as well as manual and automated test execution, are used to choose trace candidates as new regression tests. To identify and pick traces, search-based algorithms and coverage measures will be employed.

2. Active model inference is used to infer new workflow models that align with the current state of the implementation from selected traces and existing workflows.

3. The modified procedures generate reduced regression test suites, which are subsequently run on the current system implementation.

4. A smart analytics and fault reporting system offers information on the system's quality based on test execution results, defects found, and development meta-data (such as changes in the code repository).

Inputs: The PHILAE approach leverages three types of system traces, each of which is a sequence of calls and responses to the web service or API under test. The three types of traces provide distinct types of information:

- *User execution traces*: result from logging the system's current (N-1) release. Many of these traces are typically available. They bring data on which operations and values are most commonly used in the real-world deployment of that release.
- *Manual testing traces*: show new API features in the next version (N), exposing new features that must be tested. However, due to cost considerations, only a few numbers of these traces are often available, insufficient for fully generated tests.

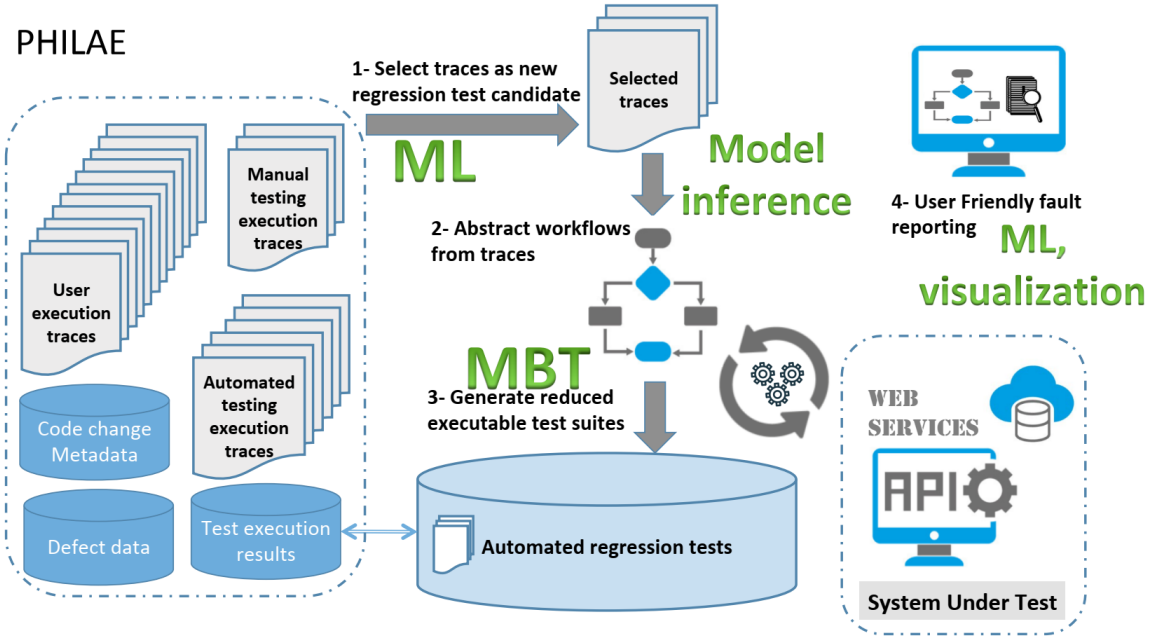


Figure 1.2: PHILAE process

- *Automated testing traces*: are created by PHILAE (from abstract workflows) to expand manual testing traces into a full regression test suite. They could also be generated by other testing tools. During the trace selection phase, some metadata from the software development process, such as code update metadata (commits), will be used. Finally, data from the defect management system will provide important information about system failures, while meta-data from the software service will provide schema and type information on the traces' data model.

The PHILAE project comprises four **research objectives**, as well as associated scientific obstacles and technical barriers that must be overcome:

PHILAE objective 1: Select trace candidates as new regression tests

This goal is to take the large set of User Execution Traces (from release N-1) and the smaller set of Manual Testing Execution Traces (from release N) and compare cluster, and prioritize them in order to select a representative sample of the traces that can be used as the basis for learning new Workflow models for Release N and generating tests for Release N.

Scientific challenges and technical barriers to overcome: one challenge in achieving this goal is learning and selecting for rare events, because while our training data will contain many traces from Release N-1 and only a few from Release N, we still want to prioritize 'rare' or 'unique' events that are new in Release N. Deep neural nets have been used in some studies in this area [Kaiser17], but we will need to discover pragmatic approaches to ensure that Release N behaviors are prioritized and learned.

PHILAE objective 2: Abstract workflows from traces

This goal will increase the abstraction level of system traces by automatically identifying and extracting high-level business workflows. This is similar to existing work on learning abstractions and automatic model inference. Still, it can be simplified in our context because inferring a set of partial models (Workflows) for diverse scenarios is adequate, rather than one entire behavioral model. These partial models can be used as a foundation for automated test generation to generate smaller automated test suites and to give limited views of system behavior. Furthermore, places where an inferred workflow differs between Release N-1 and Release N will be used as trig-

gers to produce additional tests that provide more systematic testing than exploratory testing or user interactions. It should be noted that while certain inferred processes may be able to build test sequences with oracle information (anticipated output values), others may only be able to generate generic oracles such as 'no-exceptions thrown' or 'web-service returns a status message'. Both types of test sequences will be handled by our test execution infrastructure.

Scientific challenges and technical barriers to overcome: combining the various abstraction algorithms that are already available (e.g. clustering, abstraction-learning, and model-inference), and ensuring that the resulting workflows are not too abstract. so that they can still be used to generate feasible test sequences; and that they are easily understandable to be put in the context of the process.

PHILAE objective 3: Automated regression test generation and execution
This goal will use model-based test generation approaches to produce executable robustness and regression tests from inferred workflows. It will construct robustness tests based on the gained knowledge about data frequencies and correlations and any meta-information about data kinds and ranges provided. It will also employ active learning and reinforcement learning techniques to increase the created test suite in order to thoroughly test the system. The created tests will be run on the system's updated Release N, resulting in a bigger collection of input traces that can be passed back into Step 1 to further refine the system's learned models. This iterative procedure will result in a robust collection of regression tests that can be used to test subsequent releases of the SUT (System Under Test) with minimal human intervention.

Scientific challenges and technical barriers to overcome: It is simple to produce too many tests, thus the key challenge here is prioritizing tests and limiting the entire generated regression suite so that it can be executed in an acceptable amount of time.

PHILAE objective 4: User-Friendly Fault Reporting
In this environment, anomaly identification is critical, therefore providing smart analytics of test execution outcomes is critical to assisting test engineers in focusing on the more error-prone sections of the SUT. This goal will use unsupervised machine learning methods based on clustering (hierarchical or flat clustering, depending on the nature of the distance function chosen) to group and sort test failures based on their priority level. It will also create intelligent test result visualization so that test failures can be seen overlaid on short and high-level business procedures. This will entail prioritizing anomalies, abstracting elements derived from test failures, and employing trace minimization and abstraction tools. This will be augmented by learning approaches based on associative networks, which propagate dependencies to assess the importance of bits of information. This will allow test engineers and business analysts to identify where the SUT is failing visually and textually, allowing them to determine which mistakes have priority or affect other problems.

Scientific challenges and technical barriers to overcome: We will have a diverse variety of users that wish to utilize this fault reporting system, thus no single point of view will suit them all. To address this, we want to provide different perspectives as well as certain configurable options. However, an iterative design approach with regular feedback from all types of users is required to ensure that the visualization system is usable and valuable.

To consolidate the contribution of the involved researchers into some real-world applications, the PHILAE partners gathered five case studies, each of which, was distinguished by its different software architecture, activity logging, bug concepts, and final objectives. Here are the titles of the case studies:

- *Scanette* : A supermarket item scanner
- *Orange Livebox (Telecom)* : TV and Internet box service

- *eShop* : GUI/REST API-based online shopping app
- *Keep calms* : DevOps e-Learning web application (Internal Orange)
- *Flexio* : Industrial processes

For the research reported in this thesis, we chose the first two case studies, namely, Scannette (Scanner) and Orange Livebox (Telecom), based on the supervisors' propositions, and due to the nature of these two, this thesis was directed towards specific problems to be solved and objectives to be fulfilled. We will cover the case studies, problem statements, and objectives in the following chapters and sections.

1.4 Contributions

The contribution of this thesis is as follows:

- It provides a survey of the state-of-the-art in log analysis and log mining. This includes the most recent works on machine learning in conjunction with log analysis.
- It proposes and formalizes a non-supervised ML-based generic methodology for obtaining final log mining artifacts such as log reduction, log anomaly detection, failure prediction, and root cause detection from raw log files. Figure 1.3 illustrates a top view diagram of the proposed method. The proposed method is a chain that consists of three phases: log preprocessing, model creation, and log mining task execution. Each phase has its own steps, which will be formalized in detail in chapter 3. As the structure of each software system and its logging scenario differ from one to another, each step of the proposed method must be customized to a certain degree to be able to accommodate the log files of each software and obtain the desired log mining outputs. Therefore, the rest of the contribution of this work is directly linked to two different case studies. The main distinction of the proposed methodology is its non-supervised model, which can lead to automated tools with lower human interaction. Moreover, based on the *semantics* of the system events and their mutual relations, it clusters them into some high-dimensional clusters in a conceptual space, which are called UC (Universal Cluster). UCs form a conceptual model for further log analysis and expose more hidden information from the log files, including their causal relationships.
- It realizes the proposed methodology in the Orange Livebox Telecom case study, where the software logging style deviates from conventional input-output recording. The available information is a record of the fast input events arriving at the system and a relatively low-paced record of the system status (CPU, memory, process count, etc.). This logging condition is what we will address as the *status monitoring* and is widely used when conventional logging is not possible. Status monitoring has not received enough attention, despite its broad application in log analysis. In such applications, faults will manifest themselves in some periods of anomalous behavior. We will accommodate the proposed method to process the status monitoring logs of the Orange Livebox Telecom case study and show how the ML model can be used to detect anomalous periods of time under the term of *Bug-Zones*, predict them online, and find their root causes. We expanded this study by applying the same method to *Train Ticket*, a well-known open-source benchmark, with the aim of providing a comparison basis with prospective research on the same domain.
- In the scanner case study, the proposed method is adopted to reduce enormous user trace logs to smaller ones with the same bug-triggering effect. The test suites are, in fact, the user

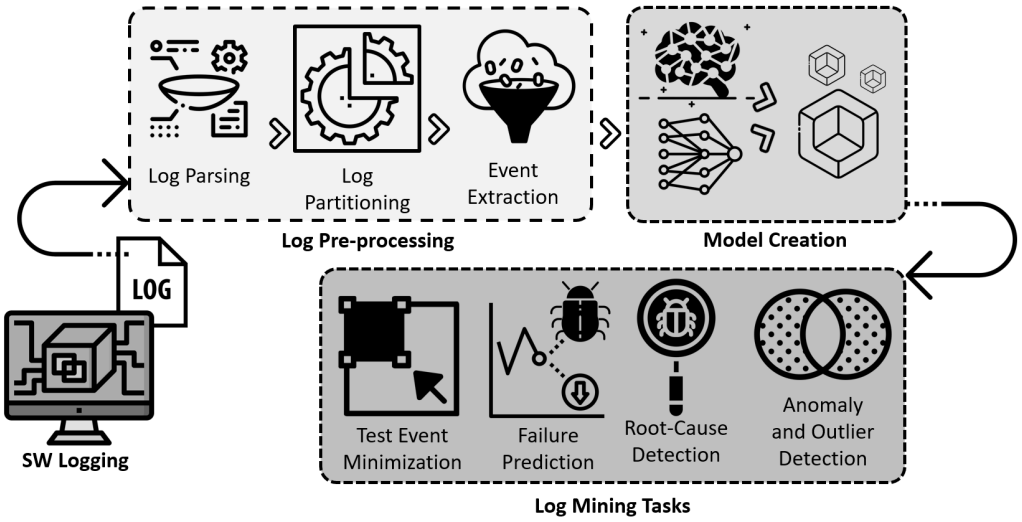


Figure 1.3: Top View of the Proposed Method

trace logs. The large test suite is able to trigger a certain number of bugs in the software system. But they are too heavy to run on the software. In this thesis, we use the proposed method to choose a tiny number of test events that represent the entire test suite and have the same or very close bug-triggering capabilities. This method saves the time and effort of running the entire usage logs and, hence, can hugely impact the costs of software testing.

- The developed chain of log processing for this research has been published as two distinct open-source tools, which are available online[4] in the PHILAE toolbox. In addition, the experiments presented in this thesis were conducted on one dataset made available as public data (Scanner case study) and another dataset from an industrial partner (Orange/ Telecom case study) that was preserved private.

1.5 Dissemination

Our research work has led to the following publications (listed in chronological order based on their publication date):

- [10]: Reducing Regression Test Suites using the Word2Vec Natural Language Processing Tool.
- [11]: Correlating Test Events With Monitoring Logs For Test Log Reduction And Anomaly Prediction.
- [12]: Telemetry-based Software Failure Prediction by Concept-space Model Creation.

1.6 Case studies

We demonstrate the practical effectiveness of the proposed method by studying three different case studies: 1) Scanner case study, 2) Orange Livebox, Telecom case study, and 3) Train Ticket benchmark case study. To achieve the second PHILAE objective, we studied the scanner case study, to test event selection from user traces, and generate a new log file. Furthermore, we worked on Telecom and Train Ticket to detect anomalies and report basic faults, as stated in Philae's objective number 4. In the following sections, there is a brief explanation of the case studies:

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

1.6.1 Scanner case study

A barcode scanner (nicknamed "Scanette" in French) is a device used for self-service checkout in supermarkets. The customers (shoppers) scan the barcodes of the items which they aim to buy while putting them in their shopping baskets. The shopping process starts when a customer (client) **unlocks** the Scanette device. Then the customer starts to **scan** the items and adds them to his/her basket. Later, customers may decide to **delete** the items and put them back in their shelves. Among the scanned items, there may be barcodes with unknown prices. In this case, the scanner adds them to the basket, and they will be processed later by the cashier, before the payment at checkout. The customer finally refers to the checkout machine for payment. From time to time, the cashier may perform a “control check” by re-scanning the items in the basket. The checkout system then **transmit**s the items list for payment. In case unknown barcodes exist in the list, the cashier controls and resolves them. The cashier has the ability to **add** or **delete** the items in the list. At the final step, the customer **abandons** the scanner by placing it on the scanner board and finalizes his purchase by **pay**ing the bill.
The Scanette system has a Java implementation for development and testing and a Web-based graphical simulator for illustration purposes. The web-based version emulates customers' shopping and self-service check-out in a supermarket by a randomized trace generator derived from a Finite-State Machine. The trace logs of the Scanette system contain interleaved actions from different customers who are shopping concurrently. Each customer has a unique session ID that distinguishes his/her traces from another customer.

To artificially inject faults, the source code of the Scanette software is mutated with 49 *mutants*, all made by a modification on the source code by hand. We needed a few logs to be used as the test bench for the proposed method. Hence, we are given three log files with different numbers of traces: 1026, 100043, and 200035. We will call them 1026-event, 100043-event, and 200035-event names, respectively. They include shopping steps for different numbers of clients (sessions). They were created as random usage logs by a generator of events that simulates the behavior of customers and cashiers. We proposed a test suite minimization approach which needs to be evaluated. We used mutation testing for evaluation purposes. The goal of the proposed TSR methods is to reduce the number of traces needed to kill the same mutants as the original test suite can kill. In the rest of this thesis, *session* and *client* are equivalent.

Overlaps with Philae
The first PHILAE's objective concerns "selecting trace candidates as new regression tests" and the third one concerns the "automated regression test generation and execution". During the regression testing, some large test suites are generated at each iteration, which in turn need to be refined to lower the testing overhead and time. This implies removing repetitive or ineffective test candidates from the new test suites in order to decrease the testing overhead. The Scanner case study, similarly, consisted of three sets of user traces of different sizes. The goal is to apply ML to refine the traces, find similar sessions, and remove the traces that drive the software through the same flow of events. In the end, the result is a reduced trace set that ideally has the same effect as the whole original set. In this case study, we are not allowed to re-run the whole trace set in order to find their effect on the software. Instead, only by observing and learning their semantics can we decide if a trace must stay or be removed. In the end, we are allowed to generate a minimal test suite and only execute the minimal test cases in order to know if it has the same effect as the large original user trace set. This condition is similar to PHILAE's objectives, in which new tests are selected to undergo regression iterations before being executed. Therefore, the achievements in this case study target the first and the third PHILAE's objectives.

1.6.2 Orange Livebox, 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 surfing, Digital TV, VoIP, Wi-Fi, P2P, Etc. All the clients' requests are recorded on their storage and accumulated later into a large log file on a daily basis (24H). Each log file is a long sequence of input events with their timestamps. Here is an example of a Test log file entry:
"timestamp": "2018-10-08T08:01:27+00:00", "metric": "loading time", "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 changes in the status information, which in turn makes the anomaly's 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.

Overlaps with PHILAE

In the Livebox Telecom case study, a long trace of system status is recorded during a long period of monitoring. During this, some system failures have occurred, caused by some tests that came into the system during the testing. We are trying to learn which sequences of events are likely to induce an anomaly. Furthermore, we wish to reduce the size of test records to assist testers in focusing on crucial time periods. This condition corresponds to PHILAE's objectives (objectives 1 and 4), which seek to identify anomalies and provide smart analytics of test execution outcomes to assist test engineers in concentrating on the more error-prone sections of the system under test (SUT).

1.6.3 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 [13]. All the microservices are related to business logic. A detailed description of the system can be found in [14]. Following [13], 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 [13], we created a simulated usage of the system (and monitored it) by running Stress-ng in a Docker server. Stress-ng [5] 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 five status system abnormal cases in the monitoring log.

Overlaps with PHILAE

We examined the Train Ticket open source benchmark to evaluate our method on a different dataset, similar to the Livebox Telecom case study. Therefore, the condition corresponds to Philae's objectives 1 and 4, which we search for to identify anomalies.

1.7 Thesis organization

The remaining chapters of this dissertation are organized as follows:

- *Chapter 2- Background and Related Works*: In chapter 2, we present the background of the concept and techniques analyzed throughout the thesis, including the review of the state-of-the-art in the field of log mining tasks.
- *Chapter 3- The Log Analysis Methodology*: chapter 3 presents a generic approach to process logs, create ML models, and extract log mining artifacts.
- *Chapter 4- Bug Prediction & Root Cause Detection: Orange Livebox - A Telecom case study*: This chapter describes the results obtained in this study by applying the proposed method to the telecom case study for software fault prediction and root-cause detection tasks.
- *Chapter 5- Log Minimization: Scanner case study*: This chapter describes the results obtained in this study by applying the proposed method to the scanner case study for test suite reduction (minimization) tasks.
- *Chapter 6- General Conclusion and Future Work*: In this chapter, a summary of the results will be given, as well as the possible future work to be made, related to this work, and the topic it presents. We also describe the problems found during this research.

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

Chapter 2

Background and Related works

Contents

2.1 Introduction

A log is a record of events that occurred during the running of a software system. Logging is done by adding lines of code to a program that, when a relevant event occurs, writes out the essential data. It is impossible to overstate the significance of event logs as a source of information in systems and network administration. Due to the ever-increasing amount and complexity of modern event logs, the task of analyzing event logs manually has become tedious [15]. For this reason, recent research has focused on the automatic analysis of these log files. There are many published works that employ logs to analyze causality and find root-cause [16, 17], cluster host [15], predict failures [18, 19], and incident diagnosis [20]. In addition, recently, machine learning and deep learning algorithms have been widely adopted by the state-of-the-art papers such as Deeplog [21] in system log anomaly detection.

In this chapter, we will explore a detailed background of the concepts studied during this thesis study, alongside relevant literature already available on the topic and that has served as the basis for this work. More specifically, in Section 2.2 we will provide a definition of Automated Log analysis to help the software testing process. We will also introduce the background of the research in the field of log mining tasks in order to provide insights into what has already been studied in this domain. In Section 2.3, on the other hand, we will describe in detail the techniques used throughout this thesis.

2.2 Automated Log Analysis For Software Testing

The system logs include a wealth of information and detail activities executed on the system. Developers and system administrators have used the system log to discover and troubleshoot system issues. Due to the increasing size of log data, log analysis automation is desirable. Machine learning methods are used extensively due to their classification and prediction abilities. Random forest, Naive Bayes, and Support Vector Machine (SVM) are highly common machine learning techniques. In recent years, due to the increase in log data size, detection, and prediction models have been trained using deep learning approaches such as RNN (Recurrent Neural Network), CNN (Convolutional Neural Networks), LSTM, and Bi-LSTM. In this section, we studied some existing works in this domain.

2.2.1 Logging

Logging is the task of constructing a logging statement with a proper description and necessary program variables and inserting the logging statements into the correct positions in the source code. The developers implement logging statements, then execute the program and collect the logs. Therefore, logging relies heavily on human expertise. They need to make informed decisions on where to log, and what to log in their logging practices during development. Logs are valuable for investigating, diagnosing, and predicting failures. According to a survey [22] involving 54 experienced developers in Microsoft, almost all the participants agreed that “logs are a primary source for problem diagnosis” and “logging is important in system development and maintenance“.

After collecting logs from executing logging statements, logs are stored mostly in CSV or JSON format. Units of information in a log are often called events, corresponding to the fact that this information is usually issued when an event is triggered. In general, events can be triggered by a user or some other external input or internally by conditions within the program. Several test cases make a test suite, which can be used to manage and execute the test cases together. An example of a log is shown in Figure 2.1.

```
172, 1584454657750, client9, scan9_2, scanner, [7640164630021], -2
173, 1584454657822, client9, scan9_2, scanner, [3046920010856], 0
174, 1584454657873, client11, scan11_0, scanner, [8718309259938], 0
175, 1584454657889, client10, scan10_3, scanner, [3520115810259], 0
176, 1584454657991, client8, scan8_1, transmission, [caisse8_0], 0
177, 1584454657991, client8, scan8_1, abandon, [], ?
178, 1584454658005, client12, scan12_1, debloquer, [], 0
179, 1584454658006, client8, caisse8_0, ouvrirSession, [], 0
180, 1584454658012, client12, scan12_1, scanner, [3474377910724], 0
181, 1584454658023, client8, caisse8_0, ajouter, [3570590109324], 0
182, 1584454658028, client12, scan12_1, scanner, [5410188006711], 0
183, 1584454658040, client8, caisse8_0, fermerSession, [], 0
184, 1584454658044, client12, scan12_1, scanner, [3046920010856], 0
185, 1584454658055, client8, caisse8_0, payer, [24.66], 0
186, 1584454658059, client12, scan12_1, scanner, [3017620402678], 0
187, 1584454658074, client12, scan12_1, scanner, [45496420598], 0
188, 1584454658116, client9, scan9_2, transmission, [caisse9_1], 0
189, 1584454658116, client9, scan9_2, abandon, [], ?
190, 1584454658119, client10, scan10_3, transmission, [caisse10_3], 1
191, 1584454658123, client12, scan12_1, scanner, [3570590109324], -2
192, 1584454658139, client13, scan13_2, debloquer, [], 0
193, 1584454658140, client9, caisse9_1, ouvrirSession, [], 0
```

Figure 2.1: A snippet of log.

2.2.2 Log Pre-Processing

The generated log is a massive amount of duplicate, unstructured, incomplete, and unclear data. We must first process this data in order to use it more effectively. There are two processes in this preprocessing. The parsing process, which transforms a raw, unstructured log into a structured log by eliminating noise and duplicate data, is the first phase. Following data abstraction, log analysis is used to group together messages of a similar type. A variety of Automatic Log Abstraction Techniques (ALATs) are available to software engineers for reducing the amount of data to be analyzed. These techniques employ numerous log abstraction algorithms developed for a variety of applications, including performance improvement and anomaly detection. These techniques are useful for identifying anomalies or failures as well as for performing root cause analyses. The classifying procedure is often referred to as log mining. By examining research publications critically, it has been determined that the top categories of log preprocessing methods are *clustering*, *filtration*, and *language modeling*.

The current log analysis methods call for improvements in data regarding resources consumed, timestamp-related defects, valuable information, and necessary to spot issues in created logs [23]. According to some researchers, the generated log contains flaws such as unsuitable log messages, missing logging statements, an insufficient log level, configuration problems with the log library, runtime problems, excessive logs, and log library updates [24]. Consequently, it is essential to assess the quality of the log data before choosing it for preprocessing. According to the authors in [25], decoding can be used to extract the relevant information from a massive supercomputer log and make it understandable. Even so, it might lead to the loss of important data. The authors in [26, 27] used conjunctive, disjunctive, and Markovic data filtering algorithms to reduce hard disk log data by 30% to 50% while taking the utility of log messages into account. Log abstraction converts raw log data into information of a higher level. Hence, log abstraction allows software engineers to apply further analyses. According to [28], the log abstraction will aid in identifying the failure's primary cause, creating a correlation between logs, and categorizing different failures. The performance of 17 ALATs (Automatic Log Abstraction Techniques) is examined in [29] based on seven quality aspects, including mode, coverage, delimiter independence, efficiency, scalability, system knowledge independence, and parameter tuning effort. The authors in [30, 31] came to the conclusion that simple NLP techniques, which are more effective than the rule-based approach, can also be used for log parsing. In order to handle complex logs and produce meaningful results, one can also concentrate on advanced NLP approaches. One of the crucial methods for log preprocessing, according to authors [32, 33], is turning the log data into time series data. In order to find event blocks that can aid in behavior analysis based on events, the researchers [34, 35] used a framework called System Log Event Block Detection (SLEBD). The Latent Dirichlet allo-

cation algorithm was used in the study by [36] to find latent topics in messages of ALMA telescope system log events. To minimize the dimension of resource usage data and visualize it at the node-specific level, [37] suggested a dynamic matrix factorization approach (dynamic MF). Anomaly identification will be aided by size reduction and straightforward visualization. The authors of the study [38] tag virtual machine log data using the C programming language. This Syslog tag is used for log classification or correlation, which aids in the failure's identification.

2.2.3 Log Mining Tasks

Log mining employs statistics, data mining, and machine learning techniques for automatically exploring and analyzing a large volume of log data to glean meaningful patterns and informative trends. The extracted patterns and knowledge could guide and facilitate monitoring, administering, and troubleshooting of software systems [39]. Furthermore, the extracted information could be used to predict future failure. Due to the high complexity of software systems, failures could come from various sources of software and hardware issues. Examples include software bugs, hardware damage, OS crash, other software, etc. In addition, on time, pinpointing the root cause by inspecting logs relies on engineers' expertise and experience. However, such information is often not well organized and documented. Therefore, complicated ways to conduct automatic log mining are in high demand. This section introduces four important log mining problems, including User Behavior Clustering, Log Minimization, Software Faults Prediction, Root Cause Analysis(RCA), and Log Anomaly Detection and we review some related work for them.

2.2.3.1 User Behavior clustering

Recently, most software systems collect activity logs that can be organized in user traces, which represent the behavior of users on those software systems. Business information systems, such as customer relationship management (CRM) and enterprise resource planning (ERP) systems, support a variety of corporate processes and enable users to carry out a wide range of tasks in an integrated and flexible way [40]. Although such flexibility is very advantageous to users, it makes businesses and software suppliers lose track of how users actually utilize these programs to carry out their work. Another example is the necessity of analyzing user behavior characteristics in a complex power grid environment in order to plan user behavior, optimize resource coordination and ultimately improve the efficiency of electricity consumption. In a different situation With the increasingly fierce market competition, capturing market demand changes and improving customer satisfaction is the cornerstone of the company to survive in the competition [41]. By dividing customers with similar requirements into separate groups, customer segmentation provides a crucial reference for companies to understand customers' requirements and develop accurate marketing programs.

These user behavior insights could be very helpful for tasks like task automation [6] which tries to identify and automate repetitive user actions, and usability engineering [7], which analyzes how software is used and may be improved. These details can be discovered by reviewing data on user interactions with software application user interfaces (UI). We shall refer to this type of data-driven analysis as "User Behavior Mining" in this study (UBM). So-called UI logs, which document event sequences as they are carried out by a user, serve as the foundation for UBM. These series of events, known as traces, each represent a simple user action in a software program, like clicking a button or typing a string into a text box. UI logs can be analyzed using methods from the field of process mining, such as applying process discovery to create a process model as a visual representation of the observed user behavior [8]. Yet, it is difficult to directly acquire specific insights due to the complexity of UI logs, which is reflected in their size and behavioral

variance. Process models found in UI logs are frequently "spaghetti" models, or extremely complicated models that are difficult for people to comprehend and analyze because of their large number of nodes and crossing edges [42, 43]. As a result, they frequently do not offer insightful information about a process.

Clustering categorizes user traces from the System Under Test (SUT) into groups based on similar behavior. Using autoencoder to reduce dimensions and then doing clustering analysis provides an acceptable solution to this issue [41]. We chose UBM as one of the log mining goals of this research because we found informative clusters of users during the analysis of log files in our case studies. Since ML-based UBM has received a great deal of attention [9] recently, we decided to investigate this task in our log analysis.

2.2.3.2 Software Failure Prediction

The user or administrator can take corrective action and alert failure situations if he or she receives information and specifics about the failure before it occurs. Finding deviations from the typical system behavior is useful for failure identification. Finding the origin of the problem and getting information on the location and timing is essential for handling failure. Once the problem has been identified, it is possible to fix it by completing the appropriate steps. Therefore, failure prediction aims to generate proactive early warnings to avoid failures, which frequently lead to unrecoverable states. Especially for large-scale software systems, the failure could have unforeseen consequences. The traditional approaches to failure management are mostly passive, which deal with it after the occurrence, while failure prediction aims to predict the failure before it happens [39].

Statistical and Machine Learning (ML) techniques have been employed in most studies to predict the error-prone modules of the software. Therefore, effective prediction of error-prone software modules can enable direct test efforts and reduce costs, help to manage resources more efficiently, and be useful for software developers.

The overview of the research articles analyzed for this literature analysis on failure prevention is shown in Figure 2.2. To increase the dependability and availability of software components, researchers have taken into account a variety of systems from the literature. The research elements that researchers use, such as systems, the kind of log data, the methodology for delivering results, and pertinent insights, are further described in Figure 2.2. These components explain the key points from a number of research articles that can significantly advance future research.

In [44] used a genetic algorithm on the RAS log to pinpoint the failure's location in the IBM Blue Gene/P system with a lead time of 0 to 600 seconds. In the product grid, the study of [50] revealed the failure pattern that supports work failure prediction. In [65], the authors employed activity log mining to discover a link between job failure and workload parameters. They concentrated their efforts on a deep learning methodology to produce early failure warning signals in the cluster of web servers and mailers. The authors of the paper [48] demonstrated a correlation between described behavior by using data mining techniques to extract the pattern in log data. A hybrid strategy produces superior outcomes compared to a single program. To automatically mine logs and give data and correlations in network failures, Zargarian et al. employed an unsupervised machine learning approach. They dealt with an actual use case processing more than 2 million alarms produced by the TIM Network Operations Center in Northern Italy over the course of two months. The majority of the attributes are categorical, necessitating the use of certain processing approaches. They decided to use frequent item rule mining. To extract temporal-spatial correlations and co-occurrences, or scenarios, they concentrate on event logs and use rule mining techniques. The authors provide visualization tools in both the spatial and temporal dimensions

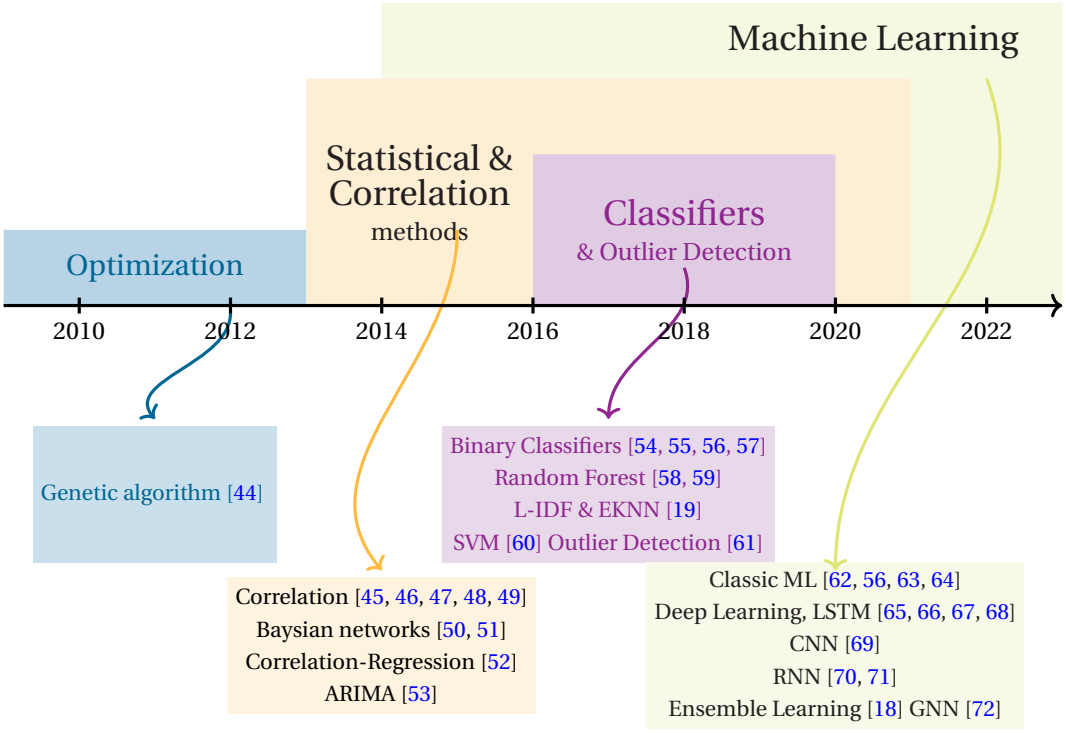


Figure 2.2: An overview of software failure prediction works based on log mining

and highlight the most crucial rules to make the analyst's job easier. Findings have been confirmed to be useful for recognizing typical circumstances and identifying potential future aberrations.

The authors of [64] come to the conclusion that the supervised closest neighbor approach outperforms unsupervised algorithms in disk failure prediction by 7%. Using supervised machine learning methods, the authors of [63] examine their work on patterns of log messages and trouble ticket data to anticipate network failures. According to the authors' 2019 research [18], ensemble learning surpasses the individual classification algorithm. The hidden Markov model approach was tested by the authors of [73] as one of the tools to assess and predict failures in a Hadoop cluster with 91% accuracy for two days in advance. A failure propagation path will help more prevent failure circumstances in fast-changing systems and failure prediction, claim [62]. Time series techniques to forecast potential failure spots in a virtual machine have been proposed in [53].

Furthermore, by identifying the type of node failure and its underlying cause, this approach can be helpful for dynamic fault tolerance [59]. According to research published in [74], the quality of software can be increased by using the right hardware and software. Several pieces of hardware in the system support online failure prediction rather than just one.

In [57] Zhou et al. proposed MEPFL to narrow down the scope of failure prediction into microservice level. MEPFL learns from system trace logs. It trains prediction models at both the trace level and microservice level to predict three common types of microservice application faults: multi-instance faults, configuration faults, and asynchronous interaction faults, based on a set of manually selected features defined on the system trace logs. MEPFL uses system trace logs collected from automatic executions of the target application and its faulty versions generated by fault injection. Semi-automatically, they inject a certain type of fault into a specific microservice for each faulty version. MEPFL can be used not just to locate faults, but also to detect failures by predicting latent errors triggered by faults. MEPFL supports four prediction models consisting of one binary classification model to classify the trace instance into two classes(with error or not), two multi-label classification model that predicts one or multiple microservices and fault types, and a single-label classification model.

According to the reviewed literature, predicting events in an HPC system can be done by observing behavior. A machine-learning technique was suggested by Pitakrat *et al.* in 2015 [62] to detect the pattern of events that frequently occur together based on the previous work of [49], which took into account log at various time periods. Recurrent Neural Network (RNN) techniques were another direction of research for fault prediction. For instance, *Seq2seq* was targeted by [70] to forecast an event that results in IoT node failure in a chosen time range. Lin *et al.* introduce a novel semi-supervised technique [71] that captures dependencies between network time series and across time points to produce meaningful representations of network activity for predicting abnormal events. The method may use the limited labeled data to explicitly learn separable embedding space for normal and abnormal samples, while efficiently using unlabeled data to deal with the lack of training data. *Desh* (Deep Learning for System Health) [67] is another RNN-based approach to forecast node failures in supercomputing systems using long short-term memory (LSTM) networks that employ RNNs. Desh discovers failure signs with training and categorization for generic applicability to log different software components without requiring modification. Desh employs a three-phase deep learning approach to (1) train to recognize chains of log events leading to a failure, (2) re-train chain recognition of events supplemented with expected lead times to failure, and (3) predict lead times during testing/inference deployment to predict which specific node will fail in how many minutes. Desh achieves an average lead time of 3 minutes with a minimum of 85% recall and 83% accuracy to take preventative steps on failing nodes, which might be used to shift compute to healthy nodes. PC^2A [68] aims to predict abnormal behaviors of a large-scale complex IT system. Their main challenges against developing efficient algorithms to predict and analyze their dataset are high-dimensional time series data, including ambiguity, complexity, and limited anomalous training samples. PC^2A proposes a combination of semi-supervised deep learning (LSTM), time series modeling, and graph analysis to have an unsupervised anomaly predictor.

According to the literature review, one strategy to stop hardware device failure is by anticipating the right time for maintenance. Also, the categorization of the event log and failure log, respectively, proved the ability to predict the precise maintenance times for ATMs [56] and vending machines [55].

Anomaly detection mechanisms can be employed to create a model for failure prediction, too. For instance, authors of [60], to discover and identify significant anomalies, use outlier detection methods: a feature-categorization-based hybrid anomaly detector and a correlation-based anomaly analyzer. Finally, they have created an SVM-based failure predictor to predict the category and lead time of various failures based on anomaly event sets. In the work of [61], a Network-Attached Storage (NAS) system with numerous hard disk drives (HDDs) and three sensors, including a thermal camera, a microphone, and system performance logs are examined. According to the unimodal results, the auditory and system performance models can detect temporal anomalies, whereas the thermal model can detect spatial anomalies. The multimodal results indicate that the multimodal strategy was able to detect failure indicators earlier than the auditory unimodal approach and before the actual failure occurred.

Clustering-based methods employed for bug prediction. The authors of LogFaultFlagger [19] describe strategies with the objective of catching the greatest amount of product faults while flagging the fewest log lines for inspection. They have observed that the lines of a failed test log should contain the location of a log error. In contrast, a passing test log should not include failure-related lines. While attempting to locate the error in a failed log, lines that appear in both a passing and failing log create noise. They introduce a method in which lines that appear in the passing log are removed from the failing log. After deleting these lines, they apply information retrieval algorithms to identify the most likely lines for further examination. The authors tweak TF-IDF to

identify the most pertinent log lines associated with previous product failures, vectorize the logs, and construct an exclusive version of KNN to determine which logs are likely to lead to product defects and which lines are the most likely indicator of the failure. LogFaultFlagger identifies 89% of all defects and less than 1% of all failed log lines for inspection. Some research investigations [58] make use of self-monitoring, analysis and reporting technology (SMART) attribute classification with Bayesian network and random forest algorithms to determine the hard disk's remaining usable time. In addition, their research reveals that SMART metrics can be used to assess the hard disk's health [75]. To decrease grid system downtime, it is possible to estimate future jobs using data mining [50] and machine learning [54].

Some other work also focused on anomaly detection and prevention for security reasons. A graph-based security analytics framework for anomaly identification and prediction is called PredictDeep [72]. The suggested approach combines graph analytics and deep learning with log data gathered from monitoring systems to add intelligence for identifying and forecasting both known and unidentified patterns of security problems. It creates a graph model out of the gathered data and depicts it. The analytical processes and their relationships are captured by the graph model. In this way, a model like this offers a knowledgeable perception of the monitored application, comprehending its activity, and spotting odd trends.

2.2.3.3 Root Cause Analysis(RCA)

Before starting this subsection, we must recall and distinguish between root-cause detection in incoming events and root-cause detection in software source code. In this study, the objective of RCA is to identify the relationship between incoming events (such as inputs, network requests, new connections, new user logins, etc.) and any anomalous software behavior. After conducting this research, for instance, we might discover that the failure happens following a certain remote user login attempt. This is the first stage of software testing, which aims to identify the action (or series of actions) that results in abnormal behavior or failure. A second stage of source code analysis, carried out by software developers, may then follow; however, the focus of this thesis is on the first stage. We chose Root-Cause Event Detection as the name of this subchapter for that reason.

Continuous integration and deployment in today's agile software development environments call for a large number of tests to be run in brief sprints [76]. The testing engineers must evaluate the enormous quantity of unprocessed diagnostics (log data) produced by these in order to find the failed tests and determine the causes of them. The method for doing this is called root cause analysis (RCA) [77], and it is typically done manually. Testing engineers typically rely on their knowledge to examine log files that they believe to be suspect. As for large-scale distributed systems, it is crucial to efficiently diagnose the root causes of incidents to maintain high system availability [78] and developers impose a significant effort to identify the cause of system failures. A large number of studies describe an approach for automatically analyzing log files and retrieving important information to determine failure causes [79, 80, 16, 81].

Software-analysis-based RCA surpasses manual RCA in terms of speed and cost because manual RCA is just too slow or expensive for the large amounts of log data being created at high speed, as well as because of its unstructured formatting, which is difficult for humans to read and understand [82]. Current methods of performing RCA based on software analysis can be categorized into *Rule-based*, *decision-tree*, *relation-mining*, and *ML-based* approaches.

A rule-based strategy uses heuristic rules depending on the expertise of the operators. a rule-based system whose behavior on the log files is entirely contained by its programmed requirements [83]. Timestamps and some significant variables in log messages are used in the construc-

tion of heuristic rules. This method works well in a relatively small system, but it is challenging to apply to other systems, such as heterogeneous network setups. Moreover, distributed systems cannot ensure the correctness of timestamps, therefore identified causality may include pseudo causality (i.e., correlation).

A decision-tree-based strategy mines the relationships between several logs to identify the precise root cause of events. To create dependency graphs and determine causality in log messages, a number of decision-tree-based techniques are used [84, 85, 86]. This method's primary flaw is that it needs a lot of log data to mine dependencies.

The relation-mining approach uses statistical measures, such as transfer entropy [87], confidence score [88], and Pearson correlation [79], to describe the link between two log time series. Depending on the metrics' value, this technique prunes unrelated edges from the initial complete graph. Because the metrics do not take causality in a theoretical sense, these methods, however, may discover erroneous causality. In this category, a few published works pursue the causality graphs for RCA [89, 33]. The causal approach has the advantage of using a theoretical causality measure rather than erroneous ones, which would naturally eliminate the impact of correlation or co-occurrence.

Based on anomaly detection and pattern matching, the authors of [16] propose *PatternMatcher* for detecting root-cause metrics. Anomaly pattern classification, which aims to filter out insignificant anomaly patterns, coarse-grained anomaly detection, and root-cause metric ranking are the three processes that make up *PatternMatcher*.

Machine-learning-based approaches allows the concerned computer to learn from the data without being constrained by rigid rules, can lead to a more reasonable, efficient, and cost-effective solution to this problem [90, 91, 92, 93, 94] since large software projects with numerous conditions and statements that must be explicitly programmed do not scale well with other methods. An additional advantage of ML-based systems is that they may potentially uncover underlying patterns that weren't initially considered by rule-based systems or manual analysis.

Related studies, mostly, require a supervised machine learning (ML) solution that can train efficiently on a limited data set in order to generate reliable root cause reports and incident predictions across varied environments. For instance, the authors of [80] employ machine learning to automate root cause analysis in agile software testing settings. For instance, after speaking with testing engineers, they extract pertinent information from raw log data (human experts). The unlabeled data are initially clustered, and although discovering modest correlations between a few clusters and failure root causes, the ambiguity in the other clusters prompts the idea of labeling. Five ground-truth categories are developed as a result of new interviews with testing engineers. They trained artificial neural networks that either classify the data or pre-process it for clustering using manually labeled data.

As authors in [17] describe, automatic techniques can be grouped into three main categories: specification-based techniques, expert systems, and heuristic based techniques which are described below:

- Specification-based techniques:
  Specification based techniques compare events logs with formal specifications that describe acceptable event sequences. The recognized anomalies are presented to testers. Unfortunately, creating and managing complete and accurate specifications is expensive and in some studies impossible. Therefore, these techniques are rarely applicable.

- Expert systems techniques:
  Expert systems have become known as one of the most interesting applications in the field of artificial intelligence. Expert systems require user-defined catalogs that describe the events

that are often associated with failures. Unfortunately, since expert systems require user-defined regular expressions to analyze and identify interesting event types, it is an expensive technique to work with catalogs [95].

- Heuristic based techniques:
  Methods focus on heuristics provide the most generic solution with the least implementation work. By using supervised and unsupervised machine learning methods, these techniques identify abnormal and normal event sequences. The supervised algorithm first analyzes the normal and abnormal event sequences and during the learning phase, the expert must distinguish between normal and abnormal executions. In contrast, unsupervised algorithms can automatically detect abnormal and normal event sequences. For example, authors in [17] combine automata learning and data clustering approaches to analyze various log file formats and find the problems. High automation is the key advantage of heuristic-based approaches, although the expressiveness of learned models limits their effectiveness.

  The technique presented in [17] performs three major tasks: event detection, data transformation, and model inference. In the event detection step, a parser is used to read the initial log file and produce a new version of the log file where an event and its attributes are stored in a single line in string format. Then the splitter refines the log file according to the granularity selected for the analysis. They used the Simple Log file Clustering Tool(SLCT) [96]. This tool identifies prefixes that frequently appear in logged events and generate a set of regular expressions that specify how to separate the constant part from the variable part. When a failure is detected, the associated log is retrieved and compared to the inferred model. Consequently, suspicious subsequences are shown to testers along with a representation of acceptable behaviors. These inputs are used by testers to determine the location and cause of failures.

While there have been various published papers on this domain using previous techniques, in recent years the techniques often fall into the deep learning category. There have been various efforts to use long short-term memory (LSTM) for anomaly identification and root-cause detection [81]. In [81] the authors used existing log anomaly detection techniques, mainly DeepLog [21] to obtain the anomaly score of the system. They aligned anomaly score and monitoring metrics and then conducted the correlation analysis based on Mutual Information (MI) to identify the root-cause metric.

Several studies have shown that they outperform traditional methods. However, unlike the statistical methods described earlier, training Deep Learning requires a significant amount of processing power. Numerous data scientists utilize expensive GPU instances to train models more quickly but at a considerable cost. If we had to train the model on each unique environment individually and continually over time, this would be an extremely expensive method for autonomously detecting incidents. Therefore, this method for monitoring logs is not suggested.

2.2.3.4 Log Anomaly Detection (LAD)

As noted by Westland [97], untreated problems become more expensive as software projects advance. There is evidence that fault identification and correction is one of the elements that most affect budget overruns. Therefore, log anomaly detection (LAD), as a step in automated log analysis, has attracted a lot of interest due to its importance in software testing and reliability engineering [39].

Finding system anomalous patterns that do not match predicted behaviors on log data is the task of anomaly detection. Common abnormalities frequently point to potential flaws, errors, or

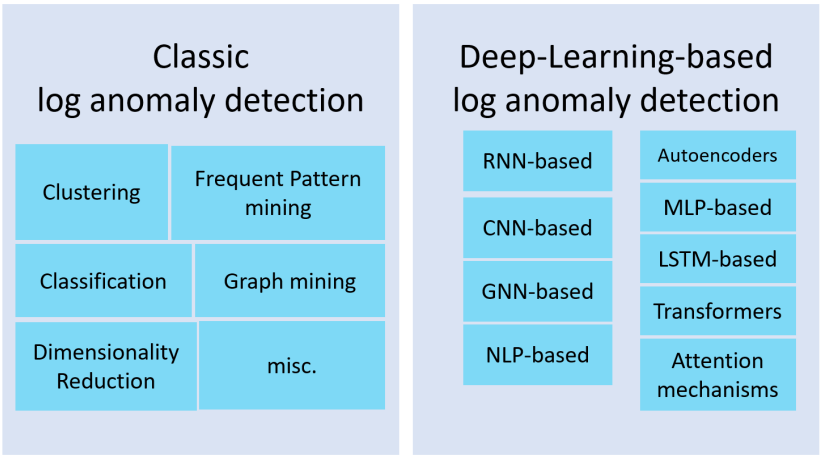


Figure 2.3: Log anomaly detection methods

failures in software systems. We can expand on the research that has already been done in this area. They divide the methods into two major categories, classic machine learning algorithms, and deep learning models, as seen in Figure 2.3.

Table 2.1 includes a list of the surveyed methodologies along with a number of key attributes. We specifically list each approach's algorithm/model, feature, and whether or not it is unsupervised and online. Unsupervised techniques, in particular, don't need labels to train models, and an online approach can handle information in real time as it comes in.

Table 2.1: Related work.

	Reference	Year	Method/Model	Unsupervised	online
Classical log anomaly detection	Kim *et al.* [98]	2020	PCA	Yes	No
	Xu *et al.* [99]	2009	PCA	Yes	No
	Lin *et al.* [100]	2016	Clustering	Yes	No
	He *et al.* [101]	2018	Clustering	Yes	No
	Liang *et al.* [102]	2007	SVM	No	No
	Kimura *et al.* [63]	2019	SVM	No	No
	Xu *et al.* [103]	2009	Frequent pattern mining	Yes	Yes
	Lou *et al.* [104]	2010	Frequent pattern mining	Yes	No
	Farshchi *et al.* [52]	2015	Frequent pattern mining	Yes	No
	Nandi *et al.* [105]	2016	Graph mining	Yes	No
	Lou *et al.* [106]	2010	Graph mining	Yes	No
	LogFlash: Jia *et al.* [107]	2021	Graph mining	No	Yes
	Yamanishi *et al.* [108]	2005	Statistical model	Yes	No
	He *et al.* [109]	2016	Logistic regression	No	No
	Micro2Vec: Cinque et al. [110]	2022		Yes	No
Deep learning	DeepLog: Du et al. [21]	2017	LSTM model	Yes	Yes
	Zhang et al. [111]	2019	LSTM classification model	No	No
	Meng et al. [66]	2019	LSTM model	Yes	Yes
	Loggan: Xia et al. [112]	2021	LSTM-based GAN model	Yes	Yes
	Lu et al. [113]	2018	CNN model	No	No

Continued on next page

Table 2.1 – continued from previous page

	Reference	Year	Method/Model	Unsupervised	online
	Log2Vec: Liu et al. [114]	2019	Graph embedding model	Yes	No
	SwissLog: Li et al. [115]	2020	Lang. Mod./Attent. BiLSTM	Yes	Yes
	LogEvent2vec: Wang et al. [116]	2020	Language Modeling (NLP)	No	No
	LogBert: Guo et al. [117]	2021	Language Modeling	Yes	Yes
	PLELpg: Yang et al. [118]	2021	Attention GRU	semi	No
	MDFULog: Li et al. [119]	2021	Lang. Mod./Attent. BiLSTM	Yes	No
	Ryciak et al. [120]	2022	Lang. Mod. (NLP)	No	No
	LayerLog: Zhang et al. [121]	2023	Lang. Mod.	Yes	No
	PULL: Wittkopp et al. [122]	2023	Lang. Mod./Attention	Yes	No

Classic machine learning LAD algorithms: Typically, classic machine learning algorithms work on top of features that practitioners expressly provide, such the log event count vector. The anomaly detection problem, in particular, can be defined into many types and solved using various methods, including clustering, classification, regression, etc.

Dimensionality Reduction: It converts high-dimensional data into a low-dimensional representation, while preserving some important characteristics of the original data in the low-dimension space. Of these algorithms, Principal Components Analysis (PCA) is one of the most widely used. If the projected distance is greater than a threshold, anomalies can be found by projecting data points to the first k main components. PCA was initially used by Xu *et al.* [99, 103] to mine console logs for system issues. In particular, a PCA model is built and fed with a log event count vector and parameter value vector for anomaly identification. To focus on root-cause analysis, Kim et al. [98] present an unsupervised anomaly detection method. Their proposed method consists of three steps: dimensionality reduction via feature selection to determine the smallest set of features that provide the most information about the network state; the use of PCA as a multivariate unsupervised learning technique to detect anomalies with low detection latency; and root cause analyses via finite state machines. They use PCA on the normal data to locate the subspace with the smallest variance of the normal data for anomaly detection. They construct a boundary of the normal data and develop an anomaly detection model based on it by defining the variation of the normal data in the subspace. Once the anomalies are identified, the message patterns of the anomaly data are compared to those of the normal data to establish where the problems are occurring. Furthermore, they investigate the error codes in the anomaly data to better understand the underlying issues.

Supervised LAD: The log partition is classified into normal or anomalous types using anomaly detection with classification, where anomalous cases differ from normal ones in terms of certain statistical features. Log anomaly detection frequently use the supervised classification technique Support Vector Machine (SVM). In [102], Liang *et al.* vectorized log partitions by identifying six different sorts of features, such as the total number of events, the number of events that occurred over time, etc. Based on these features, they applied four classification models for anomaly detection, i.e. including SVM and closest neighbor predictor. In addition, Kimura *et al.* [63] developed a log analysis approach based on the properties of logs, such as frequency, periodicity, burstiness, and association with maintenance and failures, for proactive failure identification. To find failures, an SVM model with a Gaussian kernel is used. A logistic regression model was used by He *et al.* [109] to identify anomalies. They chose event count vectors as the feature and used a collection of labeled data to train the model. When the logistic function's probability estimate is more than

0.5, a testing occurrence is deemed anomalous.

Unsupervised LAD: Clustering-based anomaly detection classifies log-based feature vectors into several clusters as an unsupervised technique so that vectors in the same cluster are more comparable to one another (and as dissimilar as possible to vectors from other clusters). Clusters with a small number of data instances are frequently abnormal. To help developers quickly spot potential issues, Lin *et al.* [100] created *LogCluster*, which groups log sequences and suggest a typical sequence. Calculating the cluster centroid enables the selection of the representative sequence. He et al [101]'s *Log3C* framework was also suggested as a way to include system KPIs in the detection of significant issues in service systems. They specifically suggested a cascading clustering approach to quickly group many log sequences. Finally, they employ a multivariate linear regression model to locate significant issues that result in KPI degradation.

Frequent Pattern Mining: It seeks to identify the most prevalent item sets and sub-sequences in a log record that represent the typical behaviors of the system. Anomalies are occurrences of data that do not fit the common patterns. Both the existence of particular log events and the chronological arrangement of log events can be patterned. For instance, Xu *et al.* [103] used log message sets that frequently co-occur to identify unusual execution traces in an online environment. Online pattern matching can more quickly detect benign system operations than offline methods can, balancing accuracy and efficiency. To find abnormalities or defects, other methods mine the sequential patterns in log events [104]. The idea of mining invariants among log messages for system anomaly identification was first put forth by Lou *et al.* [106] as well. Invariants in textual logs, which represent the equivalency relation, and invariants as a linear equation, which is the linear independence relation, are two types of invariants that are used to characterize the relationships between the various log messages. A similar strategy was put forth by Farshchi *et al.* [52], which mines the correlation and causation relationships between log events and changes in cloud system metrics. They used a regression-based methodology in particular to learn a collection of claims that model linear relations. Then, anomaly detection is carried out by keeping an eye on log event streams and comparing metrics compliance to the assertions.

Graph mining: The collection of methods primarily uses graphical features, or different graph models, to spot behavioral changes in complex systems. This allows for the early diagnosis of anomalies and the proactive taking of corrective measures. To discover anomalous runtime behaviors of distributed systems from execution logs, encompassing both sequence anomaly and distribution anomaly, Nandi *et al.* [105] presented a control-flow graph (CFG) mining technique. When an expected child for a parent node is absent within the specified time frame, a sequence anomaly is detected; meanwhile, a distribution anomaly is produced whenever an edge probability is broken. In [57], Fu *et al.* learned an FSA using log sequences to represent the execution behaviors of each system module as a state transition graph. The learnt FSAs have transitions that each match a log key. The time spent and the circulation number is collected for each state change and used to identify two performance issues: a low transition time and a low transition loop. A suitable threshold can be established to automatically identify low-performance transitions when using the Gaussian distribution to simulate the state transition in a distributed system.

Miscellaneous statistical models: There are some algorithms that don't fit into the categories listed above. For instance, a combination of Hidden Markov Models were used by Yamanishi *et al.* [108] to monitor syslog behaviour. With an online discounting learning approach, the model is specifically trained by dynamically choosing the ideal number of mixture components. Using universal test statistics with dynamically adjustable thresholds, anomaly scores are assigned. Via the identification of a group of interconnected log event occurrences and variable values that show the greatest divergence between log sets, Nagaraj *et al.* [84] were able to diagnose performance concerns for large-scale distributed systems. To be more precise, they initially divided the logs

into two sets in accordance with some performance metrics (e.g., runtime). Then, using t-tests, they suggested logging the events or state variables most responsible for the performance difference. In 2022, the authors of [110] proposed Micro2vec, a heuristic method to mine numerical representations of computer logs, which allows inferring "actionable" correlations for anomaly identification. This method is similar to language modeling techniques. The method requires no catalogs of abnormalities symptoms, embeds no application knowledge, and makes no assumptions about the structure or semantics of the underlying logs. According to the results, evaluating metrics derived from various logs makes it easier to see anomalies, which are identified by a signature incorporating many logs. It also makes it possible to infer explicable detection criteria that are difficult to spot by human specialists. Moreover, log variants derived from regular logs can aid in the detection of genuine abnormalities and outperform one-class classifiers.

Deep Learning Models: Deep learning gradually extracts features from inputs using a multiple-layer architecture (i.e., neural networks), with different layers addressing different degrees of feature abstraction. Neural networks are frequently used in log-based anomaly detection because of their extraordinary capacity for modeling complex interactions. We divide the deep learning models into RNN, CNN, GNN, MLP, LSTM, autoencoders, transformers, and attention mechanisms as depicted in Figure 2.3.

Neural Network models: LSTM and GRU models from the RNN family are frequently employed to automatically recognize the sequential patterns in log data. When log patterns diverge from the model's expectations, anomalies are raised. For instance, Du et al. [21] presented *DeepLog*, which uses an LSTM model to predict the following log event given a sequence of previous log events in order to learn the system's typical execution patterns. Instead of deviating from a typical execution route, some anomalies, however, may appear as an abnormal parameter value. As a result, *DeepLog* also uses an LSTM model to validate the parameter value vectors. Numerous previous studies make the supposition that the collection of unique log events is fixed and well-known, and that the log data are stable over time. However, Zhang et al. [111] observed that log data typically contain previously unreported log events or log sequences, suggesting log instability. In order to solve this issue, they proposed LogRobust, which uses pre-existing word vectors to extract the semantic information of log events. A bidirectional LSTM model is then used to detect anomalies. Meng et al. [66] discovered that existing word2vec models did not effectively distinguish between synonyms and antonyms in terms of capturing the semantics of log. Therefore they specifically trained a word embedding model to take into account synonym and antonym information. Meng et al[135] .'s proposal for a semantic-aware representation paradigm for online log analysis furthered their work. Out-of-vocabulary (OOV) and log-specific word embedding problems are both addressed. Zuo et al's recent.work [123] coupled transaction-level topic modeling with learning the embedding of logs, where a transaction is a collection of logs that occur in a particular order throughout time. In order to exchange anomalous knowledge between two software systems, Chen et al. [124] used transfer learning to overcome the issue of insufficient labels. To extract sequential log features, they first trained an LSTM model on the data with enough anomaly labels, and then input those features into fully connected layers for anomaly classification. After that, the fully linked layers were repaired and the LSTM model was adjusted using logs from a different system with less labels.

Language modeling: Semantic extraction, word-embedding or language modeling is a strong trend for log anomaly detection proposed in very recent papers published in the last few years [114, 120]. They are mainly based on embedding words in log data into vectors of numbers by taking into account their occurrence distances (text adjacency, temporal, etc.). For instance, the authors of [117] proposed LogBERT, a self-supervised architecture based on their previous work on Bidirectional Encoder Representations from Transformers for log anomaly detection (BERT). They em-

ploy BERT to identify patterns in normal log sequences in response to the tool's outstanding performance in modeling sequential text data. They anticipate that by utilizing the BERT structure, the contextual embedding of each log entry will be able to collect the data of whole log sequences. They suggest two self-supervised training exercises to do this: 1) Masked log key prediction seeks to accurately predict log keys in normally occurring log sequences that are randomly masked; 2) Volume of Hypersphere Minimization seeks to minimize the distance between ordinarily occurring log sequences in the embedding space. After training, they anticipate that LogBERT will encode the knowledge of typical log sequences. Then, based on LogBERT, they design a criterion to identify abnormal log sequences.

SwissLog [115] proposed a novel time embedding approach to encode temporal information, continuing the BERT-based language processing LAD track by adding time information in logs. The next step is for Attentionn-based Bi-LSTM to learn the fixed pattern of log data using the concatenation of semantic embedding and time embedding. Lastly, if an abnormality is discovered, an alarm is generated. BERT-based track was continued by MDFULog [119] with an effort to remove noisy unstable data by employing a feature augmentation approach that completely exploits the association between the semantic, time, and sequence aspects to find different kinds of log exceptions.

In an effort to better utilize the semantics of log data, in 2023, the authors of [121] offer LayerLog, a novel framework for log sequence anomaly detection based on the hierarchical semantics of log data, which takes into account the three-layered structure of log data, known as the "Word-Log-Log Sequence" hierarchy. Execution order anomalies, operational anomalies, and incomplete log sequence anomalies can all be simultaneously detected end-to-end using LayerLog. Likewise in [122] in 2023, instead than using labeled data, the developers of PULL developed an iterative log analysis method for reactive anomaly identification based on predicted failure time windows supplied by monitoring systems. Their attention-based model implements an iterative learning technique for positive and unknown samples (PU learning) to identify anomalous logs and incorporates a novel objective function for weak supervision deep learning that takes into consideration imbalanced data. Their analysis demonstrates that PULL outperforms 10 benchmark baselines across three distinct datasets in a consistent manner and detects anomalous log messages with an F1-score of greater than 0.99 even within ambiguous failure time windows.

Several model architectures, including those based on RNNs, are important in the detection of anomalies in logs. For instance, LogGAN, an LSTM-based Generative Adversarial Network(GAN), was proposed by Xia et al. [112]. The concept of the Generative Adversarial Network (GAN) was proposed by Goodfellow et al. [125] where GAN considers a machine learning problem as a game between two models (i.e., generator and discriminator). The generator and the discriminator are both present in LogGAN, as in all GAN-style models. The discriminator seeks to separate fake instances from genuine and synthetic data, while the generator tries to capture the distribution of real training data and creates realistic examples.

The viability of using Convolutional Neural Networks (CNN) for anomaly detection is also being investigated. To be more precise, Lu et al. [113] used a word embedding technique to convert logs into two-dimension feature matrices, which were subsequently processed using CNN models with various filters to detect anomalies. An approach based on graph embedding called *log2vec* was proposed by Liu et al. [114] for the detection of cyberthreats. They specifically learned the embedding of each log entry by employing a graph representation learning approach after first converting logs into a heterogeneous graph using heuristic principles. Logs were classified into clusters based on the embedding vectors, and groups with sizes below a predetermined threshold were flagged as malicious.

Probabilistic Label Estimation or PLELog [118] is another probabilistic method proposed by

Yang *et al.*. They suggest PLELog, a novel practical log-based anomaly detection method that is semi-supervised to do away with laborious manual labeling and combines knowledge of prior anomalies via probabilistic label estimation to capitalize on the advantages of supervised approaches. Semantic embedding and attention-based GRU neural networks are used by PLELog to efficiently and effectively detect abnormalities, as well as to maintain immunity to unstable log data. The results show the effectiveness of PLELog, greatly exceeding the comparative approaches, with an average of 181.6.

2.2.3.5 Test Suit Minimization

Generally, the duration of regression testing is determined by the size of the test suites. As the size of regression testing increases, its execution becomes increasingly compute-intensive. A large test suite might take weeks to run [126]. The number of test suites grows over time, resulting in a growing amount of time spent on each test run [127]. Test Suite Minimization or Test Suite Reduction(TSR) provides more efficient and simpler test suite maintenance, which in turn reduces the cost of the software testing phase, although in terms of the ability to detect faults. These methods work by identifying and removing obsolete or redundant test cases. According to the literature, we can classify TSR techniques into some categories, the most important of which are: Coverage based, Greedy algorithm, clustering methods, and Genetic algorithm [128]. Greedy-based approaches employ one of the greedy algorithms to determine the reduced test suite based on the best strategy at the moment. Over each iteration, the greedy algorithm includes the test case with the highest greedy property, such as the highest statement coverage, to the reduced test suite, which is a locally optimal solution. When the desired percentage of coverage is attained, the process ends. Coverage-based techniques ensure that the given tests, even when reduced, cause the program to be tested to run according to most of the run paths defined for that program. The techniques of clustering categories, as the name implies, take advantage of well-known clustering techniques. The purpose of cluster analysis is to partition the population such that objects with similar attributes are grouped together. After clustering, test cases are sampled from each cluster. Since the sample is selected from each cluster containing similar test cases, the characteristics of this form of sampling minimize redundancy in the subset. In the category of Genetic algorithm, existing test cases provide the initial population for the technique which then uses mutation, crossover, and fitness functions that use the information collected after the tests have been executed (for instance, information on test coverage) to generate next populations until they find the minimum test suite.

Almost all the previous test suite reduction techniques were able to substantially reduce the size of test suites. Nonetheless, it is crucial to determine how well these reduced suites can be compared to their corresponding unreduced suites using criteria other than suite size. Since the purpose of executing test cases is to identify software faults, one metric for evaluating the quality of a test suite is its ability to detect faults.

2.3 Data Analysis and Machine Learning Techniques

Software testing automation has been adopted as a feasible strategy to avoid the complexity and expense of most testing activities. Exploiting machine learning models in software testing has received increasing attention during the last few years. This section covers some techniques on ML (Machine Learning) techniques and lays out the ML techniques that are used in this thesis.

2.3.1 Machine Learning Techniques

Over the past years, the research on artificial intelligence, more specifically machine and deep learning, has flourished. Machine learning has been successfully applied in many areas of software engineering including : behavior extraction, time series analysis, software testing, anomaly and root cause detection. In addition, there are lots of research in the software testing facets that supported by ML. For instance, test case prioritization, test case constructions and Mutation testing automation. Considering machine learning, a clustering method has been developed for selecting regression tests, while a combination of genetic algorithms and clustering has been used to select test cases in a multi-objective test-optimization setting. These works are interesting and suggest that, despite understandability and complexity issues, the usage of machine learning (clustering, reinforcement learning, etc.) in software testing is a key-enabling technology. Here, we present some ML techniques that we used in our study.

2.3.1.1 Clustering methods

K-Means: The standard version of the k-means algorithm was proposed in 1957 by Lloyd [129] in the field of signal processing for the pulse-code modulation technique (PCM), and it was later published in 1982. The name 'k-means' was first introduced in 1967 by James MacQueen in his k-means version [130]. Apart from the Lloyd's and McQueen's versions, some researchers use Forgy's algorithm [131] as the standard algorithm for the k-means implementation.

The k-means concept is used in different domains for the cluster analysis purposes as it is simple to implement and produces effective clustering results in less amount of time. In addition, the k-means algorithm is able to detect the dissimilar data values in the dataset, also called Outliers.

The PCM technique is used to map a large input set of the analog signal values into their corresponding digital values, i.e., the method is used to digitally represent the analog signals. This process is called vector quantization, where a large set of sampled analog values are divided into a small (countable) number of groups. Each of the resulted groups contains similar values, which are closer to each other. Also, each group is represented by an average of all the values of the group, termed as *centroid* of that group.

Apart from the areas of the signal processing and vector quantization, the k-means finds its applications in the diverse fields such as data compression, image processing, market segmentation, computer graphics, etc. The K-means algorithm is widely used nowadays for the cluster analysis purposes in the field of data mining. In the context of this thesis, K-means has been our primary choice wherever a clustering algorithm was needed.

There exists different versions of the k-means algorithm in today's time, which are known as the modern k-means. In the modern k-means versions, researchers apply different heuristics to refine the initial condition of partitioning the data, producing efficient results.

In K-means, each cluster is represented by its center (called a "centroid"), which corresponds to the arithmetic mean of data points assigned to the cluster. A centroid is a data point that represents the center of the cluster (the mean), and it might not necessarily be a member of the dataset. This way, the algorithm works through an iterative process until each data point is closer to its own cluster's centroid than to other clusters' centroids, minimizing intra-cluster distance at each step.

K-means searches for a predetermined number of clusters within an unlabelled dataset by using an iterative method to produce a final clustering based on the number of clusters defined by the user (represented by the variable K). For example, by setting "k" equal to 2, the dataset will be grouped in two clusters, while if we set K equal to 4 we will group the data in four clusters.

K-means triggers its process with arbitrarily chosen data points as proposed centroids of the groups and iteratively recalculates new centroids in order to converge to a final clustering of the

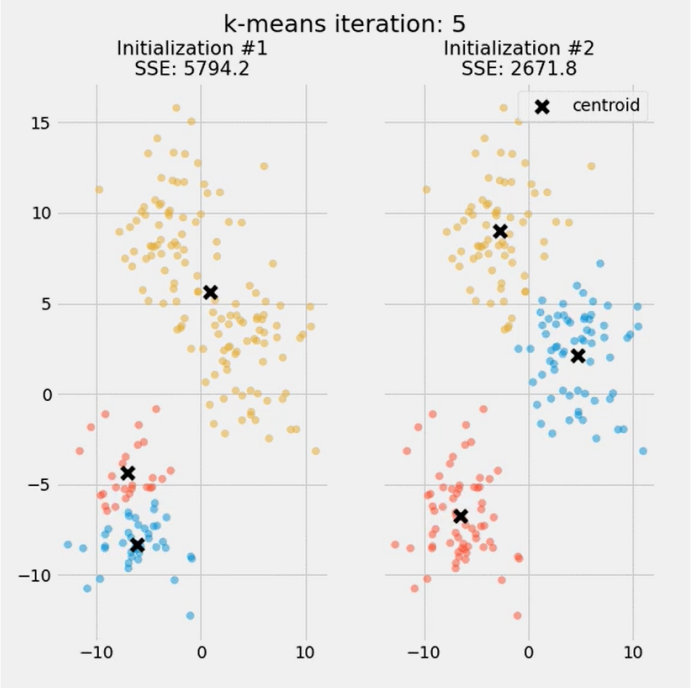


Figure 2.4: An example of K-means centroids after 5 iterations. Source: pinecone.io

data points. Specifically, the process works as follows:

The algorithm randomly chooses a centroid for each cluster. For example, if we choose a “k” of 3, the algorithm randomly picks 3 centroids. K-means assigns every data point in the dataset to the nearest centroid, meaning that a data point is considered to be in a particular cluster if it is closer to that cluster’s centroid than any other centroid. For every cluster, the algorithm recomputes the centroid by taking the average of all points in the cluster, reducing the total intra-cluster variance in relation to the previous step. Since the centroids change, the algorithm re-assigns the points to the closest centroid. The algorithm repeats the calculation of centroids and assignment of points until the sum of distances between the data points and their corresponding centroid is minimized, a maximum number of iterations is reached, or no changes in centroids value are produced. Figure 2.4 illustrates a K-means clustering including its seven centroids after five iterations.

Finding the value of K:

How do we choose the right value of “k”? One popular approach is testing different numbers of clusters and measuring the resulting Sum of Squared Errors (SSE), choosing the “k” value at which an increase will cause a very small decrease in the error sum, while a decrease will sharply increase the error sum. This point that defines the optimal number of clusters is known as the “elbow point”. During our study, we used Elbow method to obtain the number of clusters.

Spectral Clustering: Spectral clustering is a graph theory-based technique for identifying groups of nodes in a graph based on the edges linking them. The method is adaptable and can be used to cluster non-graph data as well. Several spectral clustering algorithms have been developed during the last two decades [132]. Data points are considered as nodes of a graph in spectral clustering. For employing spectral clustering, we require a robust graph that indicates the data’s similarity. There are various methods for constructing the affinity matrix. By computing a graph of the nearby neighbors, it is simplest to generate the affinity matrix. Every data point is represented as a node in a k-nearest neighbors graph. Then, an edge is drawn between each node and its k nearest neighbors in the initial space. Since it relies on the idea that "close" nodes should belong to the same cluster, it may not be applicable to all datasets. A more comprehensive strategy is to interpret data as a precomputed affinity matrix. Here, an affinity matrix corresponds to an adjacency

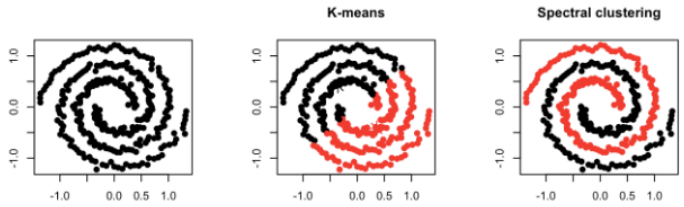


Figure 2.5: A widely-used example on the difference between K-Means and Spectral Clustering, Source: kaggle.com

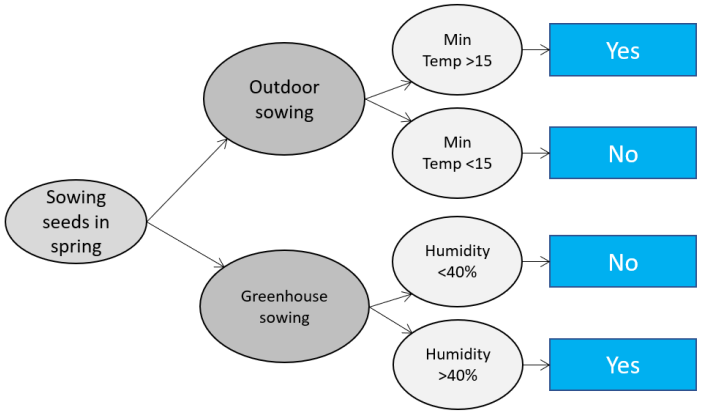


Figure 2.6: A simple decision tree on sowing seeds based on different conditions

matrix, with the difference that the value for a pair of points conveys how similar those points are. If pairings of points are very dissimilar, the affinity should be 0. If the points are the same, the affinity could be 1. In this way, the affinity corresponds to the weights of the graph's edges.

K-Means vs. Spectral Clustering:
Compactness – Nearby points fall into the same cluster and are grouped together in close proximity to the cluster's center. The separation between the observations can be used to gauge proximity. E.g.: Clustering with K-Means
Connectivity – A cluster is made up of points that are connected or located close to one another. Even though two points are closer to one another, they are not grouped together if they are not connected. This strategy is used in the process of spectral clustering.

Even when the true number of clusters K is known to the algorithm, K-means will fail to efficiently cluster them. K-means is an excellent data-clustering algorithm for identifying globular clusters in which all members of each cluster are in close proximity to one another (in Euclidean sense). Spectral clustering is more generic (and powerful) since it behaves like k-means if we only utilize Euclidean Distance in its similarity matrix. Yet, the opposite is not true. Figure 2.5 illustrates the difference of these two clustering algorithms in a visual example. We can observe how the spectral clustering can be useful to extract connectivity from the dataset. We have compared K-mean and spectral clustering algorithms in our approach in chapter 5.

2.3.1.2 Classification methods

The Classification algorithm is a technique for Supervised Learning that identifies the category of incoming observations based on training data. In this section, we will discuss some classification algorithms like Random forest and SVM (Support Vector Machine) which we used during our research.

Because the random forest model is made up of several decision trees, it would be useful to begin by briefly describing the decision tree algorithm.

Decision Tree: A decision tree is a non-parametric supervised learning approach that can be

used for classification as well as regression applications. It has a tree structure that is hierarchical and consists of a root node, branches, internal nodes, and leaf nodes. A decision tree, as shown in the Figure 2.6, begins with a root node that has no incoming branches. The root node's outgoing branches then feed into the internal nodes, also known as decision nodes. Both node types evaluate the given features to generate homogeneous subsets, which are denoted by leaf nodes or terminal nodes. The leaf nodes represent all of the dataset's conceivable outcomes. As an example, suppose we were trying to decide whether to sow seeds at the start of the season. We could use the decision criteria shown in Figure 2.6 to make a decision.

This type of flowchart layout also produces an easy-to-digest picture of decision-making, allowing various groups within an organization to better comprehend why a choice was taken. By undertaking a greedy search to determine the optimal split points inside a tree, decision tree learning applies a divide and conquer technique. This dividing process is then repeated top-down and recursively until all or the majority of entries have been categorised under particular class labels. The decision tree's complexity determines whether or not all data points are classed as homogeneous sets. Smaller trees can achieve pure leaf nodes more easily. data points in a single class. However, when a tree increases in size, maintaining this purity becomes increasingly challenging, and this usually results in too little data falling under a given subtree. This is known as data fragmentation, and it frequently leads to over-fitting. As a result, decision trees prefer small trees, which is consistent with Occam's Razor's principle of parsimony, which states that "entities should not be multiplied beyond necessity." In other words, decision trees should add complexity only if necessary, because the simplest explanation is often the best. Pruning is commonly used to minimize complexity and prevent over-fitting; this is a technique that removes branches that split on features of low value. The model's fit can then be tested using the cross-validation procedure. Another method for decision trees to preserve their accuracy is to construct an ensemble using a random forest algorithm; this classifier predicts more accurate outcomes, especially when the individual trees are uncorrelated with one another.

Random Forest: Random forest is a popular machine learning technique developed by Leo Breiman and Adele Cutler that combines the output of numerous decision trees to produce a single conclusion. Its ease of use and flexibility, as well as its ability to tackle classification and regression challenges, have boosted its popularity. The random forest algorithm is a bagging method extension that employs both bagging and feature randomness to produce an uncorrelated forest of decision trees. Randomness of features, commonly known as feature bagging or "the random subspace approach" [133], creates a random collection of characteristics, ensuring that decision trees have low correlation. This is a significant distinction between decision trees and random forests. Random forests select only a subset of the available feature splits, whereas decision trees consider all of them.

Random forest techniques have three major hyper-parameters that must be set prior to training. These variables include node size, number of trees, and number of characteristics sampled. The random forest classifier can then be used to address regression or classification problems.

The random forest algorithm is made up of a collection of decision trees, and each tree in the ensemble is made up of a bootstrap sample, which is a data sample obtained from a training set with replacement. One-third of the training sample is set aside as test data, known as the out-of-bag (oob) sample, which we'll discuss later. Another instance of randomization is then injected into the dataset using feature bagging, increasing diversity and decreasing correlation among decision trees. The forecast determination will differ depending on the type of difficulty. Individual decision trees for a regression job will be averaged; e,g: a majority vote for a classification problem. The anticipated class will be determined by the most frequent category variable. Lastly, the oob sample is used for cross-validation, which completes the prediction.

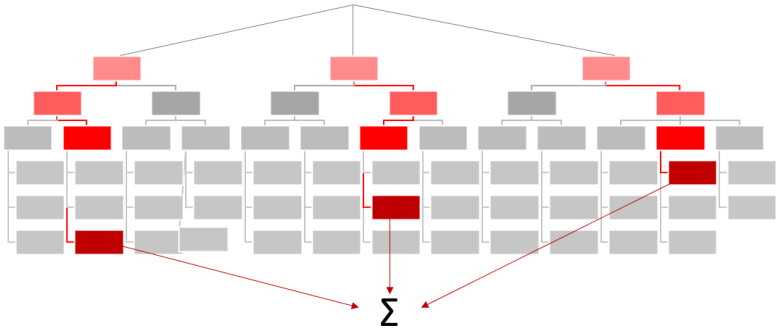


Figure 2.7: Random forest: a forest of decision trees

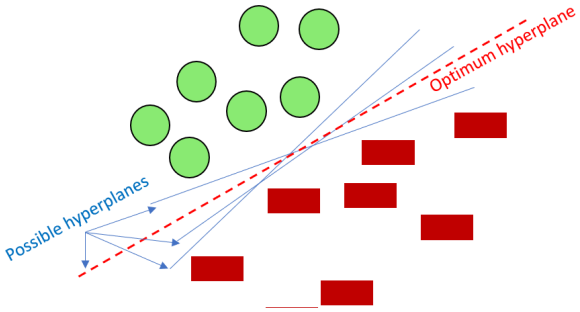


Figure 2.8: Possible and optimum hyperplanes in Support Vector Machine (SVM)

As each decision tree classifies a given data value, the random forest method is used to determine the most frequent predicted value among all decision trees and outputs this as the final predicted class of the selected data value. as shown in Figure 2.7. In our work, we used Random Forest in chapter 4. We employed it as a classifier to determine if the sequence is Pre-Bug-Zone or Random-Zone. In addition, Random Forest is used as a predictor during the on-line prediction phase. More detail will be discussed in chapter 4.

Support Vector Machine (SVM): Support Vector Machines (SVM) are one of the most well-known and discussed machine learning techniques. They were immensely popular during their development in the 1990s and remain the go-to solution for a high-performing algorithm with a little tweaking. The goal of SVM is to find a hyperplane in an N-dimensional space (N-Number of features) that classifies the data points clearly. The Support Vector Machine classifier is a generalization of the maximal margin classifier. This classifier is straightforward, but it cannot be used to the vast majority of datasets since the classes must be divided by a linear boundary.

In the context of support-vector machines, the ideally separating hyperplane or maximum-margin hyperplane is a hyperplane that is equidistant from two convex hulls of points (Figure 2.8).

Support Vectors and Hyperplanes:

A hyperplane is a flat affine subspace of dimension N-1 in an N-dimensional space. A hyperplane is represented visually as a line in 2D space and as a flat plane in 3D space Figure 2.9.

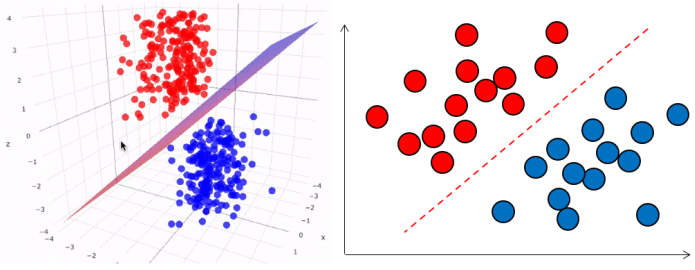


Figure 2.9: 2D and 3D hyperplanes in Support Vector Machine (SVM)

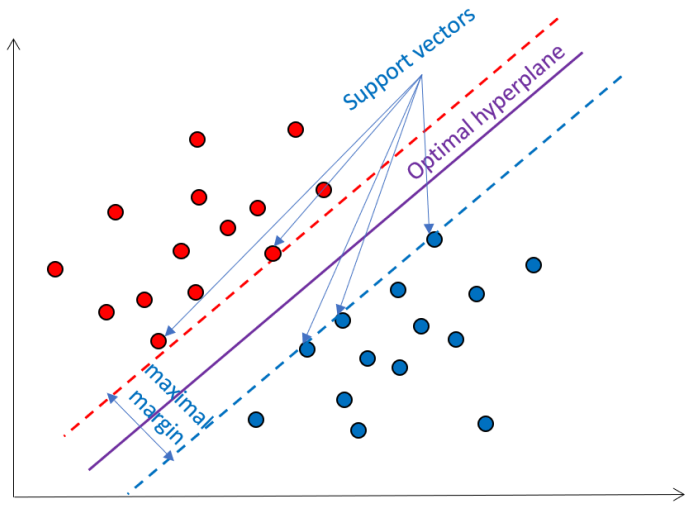


Figure 2.10: Support vectors in SVM

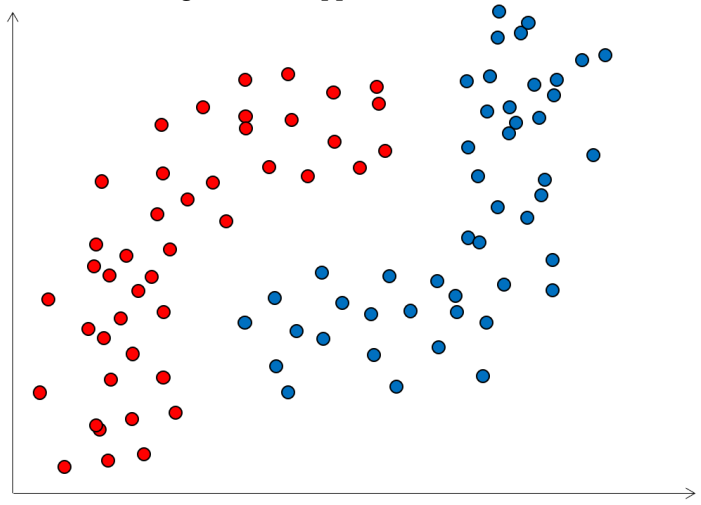


Figure 2.11: Non-linear feature space

In layman's terms, a hyperplane is a decision boundary that aids in the classification of data points. A hyperplane is a flat affine subspace of dimension N-1 in an N-dimensional space. A hyperplane is represented visually as a line in 2D space and as a flat plane in 3D space.

In layman's terms, a hyperplane is a decision boundary that aids in the classification of data points. There are numerous hyperplanes that could be used to split two classes of data points. Our goal is to locate a plane with the greatest margin, or the greatest distance between data points from both classes, as seen in Figure 2.8.

Support Vectors are data points that are on or near the hyperplane and influence the hyperplane's position and direction (Figure 2.10). Using these support vectors, we maximize the classifier's margin, and eliminating them changes the position of the hyperplane.

The hyperplane is equidistant from the Support Vectors. These are termed support vectors because as their location changes, so does the hyperplane. This means that the hyperplane is exclusively dependent on the support vectors and no additional observations. SVM, as stated thus far, can only categorize data that is linearly separable.

Non-linear SVM:

What if the data is non-linearly separated? For instance, in Figure 2.11, the data is non-linearly separated and we cannot draw a straight line to classify the data points. The notion of *Kernel* in SVM is used to categorize non-linearly separated data. A kernel is a function that converts lower-dimensional data to higher-dimensional data. A kernel function is a way for taking input data

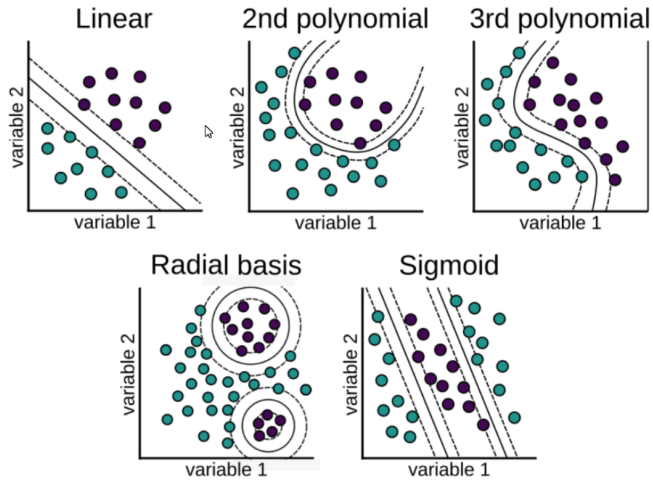


Figure 2.12: Different SVM kernels and their distinctive feature. Source:[http://people.ciirc.cvut.cz/ hlavac]

and transforming it into the needed form of processing data. The term "kernel" refers to a set of mathematical functions used in Support Vector Machine to provide a window to manipulate data. Hence, in general, the Kernel Function modifies the training set of data so that a non-linear decision surface can transform to a linear equation in a larger number of dimension spaces. It essentially returns the inner product of two points in a typical feature dimension.

$K(\bar{x}) = 1, if \|\bar{x}\| <= 1$

$K(\bar{x}) = 0$, Otherwise

Gaussian Kernel: When there is no prior knowledge about the data, the Gaussian Kernel is employed to conduct transformation.

$K(x,y) = e^{-\left(\frac{\|x-y\|^2}{2\sigma^2}\right)}$

Gaussian Kernel Radial Basis Function (RBF): The same as the previous kernel function, but with the addition of the radial basis approach to improve the transformation.

$K(x,y) = e^{-}\left(\gamma\|x-y\|^2\right)$

$K(x,x1) + K(x,x2)$(Simplified -Formula)

$K(x,x1) + K(x,x2) > 0$

$K(x,x1) + K(x,x2) = 0$

Sigmoid Kernel: This function is comparable to a two-layer neural network perceptron model, and it is utilized as an activation function for artificial neurons.

$K(x,y) = \tanh\left(\gamma \cdot x^T y + r\right)$

Polynomial Kernel: It shows the similarity of vectors in a feature space in the training set of data over polynomials of the original variables utilized in the kernel.

$K(x,y) = \tanh\left(\gamma \cdot x^T y + r\right)^d, \gamma > 0$

Figure 2.12 illustrates four different SVM kernels and their distinctive feature on different data sets.

Ensemble Methods: Ensemble learning approaches are made up of a collection of classifiers. The predictions of decision trees are pooled to determine the most popular result. Some of the advanced ensemble classifiers are: Stacking, Blending, Bagging and Boosting. Bagging, also known as bootstrap aggregation, and boosting are the most well-known ensemble approaches. Leo Breiman [134] invented the bagging method in 1996; in this method, a random sample of data from a training set is picked with replacement—that is, individual data points can be chosen more than once. Following the generation of multiple data samples, these models are trained independently and depending on the type of task—i.e. The average or majority of such predictions yields a more accurate estimate in regression or classification. This method is frequently used to reduce variance

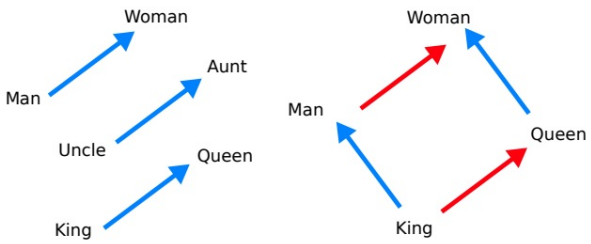


Figure 2.13: The analogy pairs in the word embedding space demonstrate surprising algebraic relationships [135].

in a noisy dataset.

2.3.1.3 Word Embedding

A word embedding is a representation of a word in natural language processing (NLP). Text analysis employs word embedding. The representation is often a real-valued vector that encodes the meaning of the word in such a way that words that are close in the vector space are assumed to have comparable meanings. Word embeddings can be obtained by language modeling and feature learning approaches, in which words or phrases from the lexicon are mapped to real-number vectors. Word embeddings are a class of techniques that represent individual words as vectors in a defined vector space. Each word is mapped to a vector, and the vector values are learned; therefore, the technique is frequently grouped with deep learning. Thus, Neural networks, dimensionality reduction on the word co-occurrence matrix, probabilistic models, explainable knowledge base technique, and explicit representation in terms of the context in which words appear are all methods for generating this mapping. In the learned vector space, words support basic algebraic operations, which is one of the surprising results of word embedding. A typical example is the study of analogy pairs - king: queen, man: woman - in which king-man plus woman equals queen (see Figure 2.13).

When employed as the underlying input representation, word embeddings have been demonstrated to improve performance in NLP tasks such as syntactic parsing and sentiment analysis. Typically, each word is represented by a vector with up to hundreds of dimensions. In comparison, limited word representations, such as a one-hot encoding, require thousands or millions of dimensions.

There are numerous neural word embedding strategies, such as word2vec [136] and Glove [137]. The following is one example of a widely used distributed text representation: Word2Vec. In this thesis, we applied Word2Vec for word embedding purposes. Therefore, we provide an explanation of it here.

Word2Vec

Word2vec is a method for computing vector representations of words developed by a team of Google researchers led by Tomas Mikolov [138, 136]. Google maintains an open-source version of Word2vec that is distributed under the Apache 2.0 license.

Word2vec is a two-layer neural network that *vectorizes* words to process text. It takes a text corpus as input and produces a set of vectors as output: feature vectors that represent words in that corpus. Word2vec is not a deep neural network, but it converts text to a numerical format that deep neural networks can understand.

The uses of Word2vec go beyond language processing. It can be applied to genes, codes, likes, playlists, social media graphs, and other verbal or symbolic sequences that include patterns. Since words, like the other data described above, are essentially discrete states, we are merely seeking for the transitional probabilities between those states: the likelihood that they will co-occur.

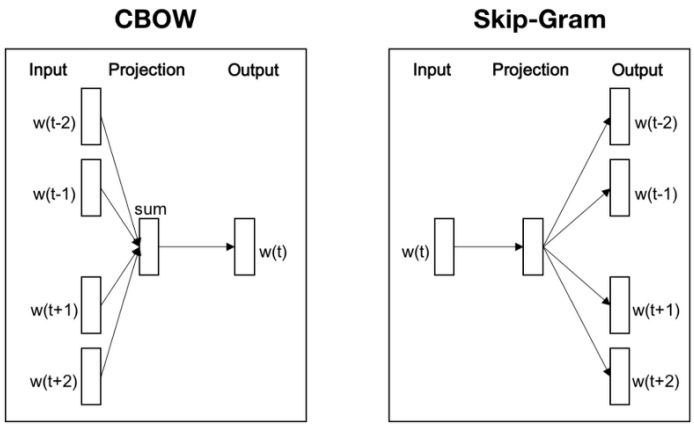


Figure 2.14: Illustration of the Skip-gram and Continuous Bag-of-Word (CBOW) models

Word2vec's objective and utility is to combine vectors of related words together in vector space. That is, it uses mathematics to find similarities. Word2vec generates vectors, which are distributed numerical representations of word characteristics, such as individual word context. It accomplishes it without the need for human involvement.

Word2vec can produce accurate assumptions about a word's meaning based on previous appearances if given enough data, usage, and circumstances. These educated estimates can be used to determine a word's relationship with other words (for example, "man" is to "boy" what "woman" is to "girl"), or to cluster and categorize texts. These clusters can serve as the foundation for search, sentiment analysis, and recommendations in disciplines ranging from scientific research to legal discovery, e-commerce, and customer relationship management.

Word2vec works similarly to an autoencoder in that it encodes each word in a vector, but instead of training against the input words via reconstruction, like a constrained Boltzmann machine does, word2vec trains words against other words in the input corpus.

Word2vec does so in one of two ways which are illustrated in Figure 2.14: it predicts a target word using context (a method known as the continuous bag of words, or CBOW), or it predicts a target context using a word (a method known as skip-gram). Therefore, skip-gram tries to guess neighboring words using the current word. The latter method is used because it yields more accurate results on huge datasets.

When the feature vector assigned to a word cannot effectively anticipate its context, the vector's components are changed. The context of each word in the corpus is a mentor, providing back error signals to alter the feature vector. By altering the numbers in the vector, the vectors of words judged similar by their context are nudged closer together.

Because the representation of each word is a vector with a size equal to the size of the vocabulary, it is typical to have (e.g.) 500 numbers arranged in a vector representing a word or group of words. Each word is represented as a point in a 500-dimensional vector pace by these numbers. More than three-dimensional spaces are difficult to visualize.

Things and thoughts that are similar are demonstrated to be "close." Their respective meanings were converted into measurable distances. Qualities are transformed into quantities, allowing algorithms to function. But, similarity is only one of the numerous correlations that Word2vec can learn. It can, for example, assess the relationships between words in one language and map them to words in another.

For instance, If we have word2vec vector of the word Paris, we can not tell much by looking at the values. However, if we visualize it a bit therefore we can compare the word Paris with other city word2vec vectors. Figure 2.15 presents countries and their capital vectors. These vectors form the foundation of a more thorough word geometry. Not only will Rome, Paris, Berlin, and Beijing

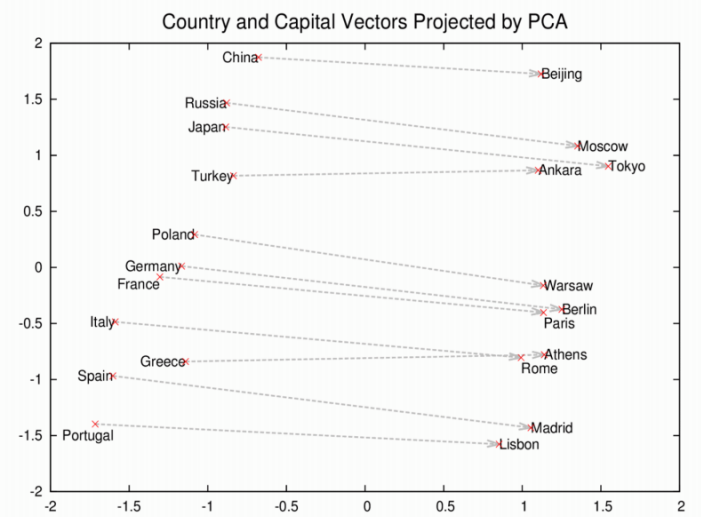


Figure 2.15: Projection of countries vectors and their capitals in 2D, learnt by Word2Vec from a text corpus

cluster together, but they will also have identical vector space distances to the countries whose capitals they are; for example, Rome - Italy = Beijing - China. And if you simply knew Rome was the capital of Italy and were curious about China's capital, the equation Rome - Italy + China would return Beijing.

Two key hyperparameters in word2vec training process are the window size and the number of negative samples. The Gensim default window size is 5 (two words before and two words after the input word, in addition to the input word itself). The number of negative samples is an another training process factor. The original paper suggests that five to twenty negative samples is an ideal ratio. It also states that a sample size of 2 to 5 is sufficient when the dataset is sufficiently large. The default for Gensim is five negative samples.

Word2vec will be employed in chapter 4 and chapter 5 in different case studies. For more details on how the input and output of Word2vec may lookalike, you can refer to these chapters.

Visualization of word embeddings

Constructing distributed representations for words using neural language models and analyzing the resulting vector spaces has become an important component of natural language processing (NLP). The NLP community has begun to adopt high-dimensional visualization techniques in order to gain insight into the relationship between words. For instance, the authors of [135] introduced new embedding techniques for visualizing semantic and syntactic analogies, along with tests to determine if the resulting views capture salient structures. Furthermore, researchers frequently utilize t-distributed stochastic neighbor embeddings (t-SNE) [139] and principal component analysis (PCA) to produce two-dimensional embeddings for assessing the general structure and studying linear relationships (e.g., word analogies), respectively.

Hence, the embedding layer has very high dimension, and it is necessary to use a technique to decrease the dimension for visualization. In particular, nonlinear dimension reduction strategies, most notably t-distributed stochastic neighbor embedding (t-SN) are used to provide a high-level overview of the embedding space. It is the most frequently utilized method for visualizing word embeddings. t-SN is optimized for two-dimensional visualization and is more likely to reveal data's inherent clusters. While, PCA, highlights word categories rather than the desired analogy direction. Mainly, It captures the largest variance in the data, which corresponds to the difference in the meaning of the words [135].

2.3.1.4 Sentence Embedding

In recent years, the concept of word embeddings has become popular in the NLP field. Due to the hierarchical nature of human languages, it is insufficient to interpret a text simply based on a comprehension of each individual word. This has led to a recent study that are semantically robust for longer segments of text, such as sentences and paragraphs. Many machine learning approaches require data in the form of a fixed-length feature vectors. When it comes to texts, one of the most common fixed-length features is bag-of-words [140, 141]. Each word is treated as an equivalent entity in the bag-of-words models [142]. A bag-of-words is a text representation that specifies the word occurrences in a document. We solely check word counts, disregarding grammatical intricacies and word order. They ignore a word's semantic meaning. A bag-of-word model, for instance, does not recognize that a horse and a pony are more similar than a horse and a dog. The TF-IDF (term frequency-inverse document frequency) method is one of the methods based on the Bag-of-words. The TF-IDF of a word within a document is computed by multiplying two metrics: The term frequency (TF) of a word in a document and The inverse document frequency(IDF) of the word across a set of documents. By multiplying the values of these two terms, the TF-IDF score of a word in a document is determined. The higher the score, the more important that word is in that document. Despite its popularity, bag-of-words techniques have two main weak points: they lose the word ordering and ignore the semantics of the words. Recent implementations, such as the doc2vec technique [141], integrate word and document embeddings to represent sequences. Mikolov introduced Doc2vec [141] as a simple extension of Word2Vec (his previous study) to learn document-level embeddings. It was proposed in two forms: dbow (distributed bag of words) and dmpv. Dbow is a simpler model that completely ignores word order, whereas dmpv has more parameters and is more complex. Dbow works in the same way as skip-gram and dmpv acts similarly to cbow.

[143] proposed two categories to compare sentence embedding approaches: non-parametrized and parametrized approaches. Parametrized sentence embedding models are parametrized and require training to optimize their parameters, whereas non-parametrized sentence embedding are parameter-free and require no further training upon pre-trained word vectors.

2.3.2 Outlier Detection Techniques

A number of different application domains employ anomaly detection techniques. Examples include Data Leakage Prevention (DLP), fraud detection, and medical applications. In each of these very different fields, synonyms are widely applied for the anomaly detection process, which includes outlier detection, novelty detection, fraud detection, intrusion detection, and behavioral analysis. Based on study in [144] Unsupervised outlier detection techniques can be categorized into four different main groups: (1) Nearest-neighbor based techniques (2) Clustering and classifier-based methods (3) Statistical algorithms and (4)Subspace techniques. Here we shortly introduce of two different methods of outlier detection techniques. Local Outlier Factor(LOF) and Isolation Forest(IF) techniques.

2.3.2.1 Local Outlier Factor

The local outlier factor (LOF) [145] is the most well-known local anomaly detection algorithm. It is categorized in the Nearest-neighbor based algorithm. LOF seeks to identify points with a lower density than their neighbors. Hence, it could detect some outliers in the low-density region overall which have even lower density compared to their neighbors, as well as some outliers in high-density regions which have a lower density compared to their neighbors. To calculate the

LOF score, three steps have to be computed: First, the K-nearest-neghbors have to be found for each record x. In case of distance tie of the K_{th} neighbor, more than K neighbors are used. The density of a record is then estimated using the k-nearest neighbors N_K by computing the local reachability density (LRD):

$$LRD_k(x) = \frac{1}{\frac{\sum_{o \in N_K(x)} d(x,o)}{|N_K(x)|}} \tag{2.1}$$

Lastly, the LOF score is determined by comparing the LRD of a record to the LRDs of its k nearest neighbors:

$$LOF(x) = \frac{\sum_{o \in N_K(x)} \frac{LRD_K(o)}{LRD_K(x)}}{|N_K(x)|} \tag{2.2}$$

In general, the LOF score is a ratio of local densities. Since it only considers its local neighborhood and the score is mainly focused on the k nearest neighbors, it is simply a ratio of local density. Obviously, global anomalies can also be discovered due to their low LRD in comparison to their neighbors. The algorithm calculates the score for each point in the dataset and utilizes LOF close to 1 as the standard to identify whether it is an outlier factor. If the ratio is closer to 1, it means that the density of x and neighbor points is not much different, and x and neighbors belong to the same cluster. If the ratio is less than 1, it means that the density of x is higher than the density of neighbors and x is dense points. if the ratio is greater than 1, it means that the density of x is less than the density of neighbors and x is an abnormal point.

2.3.2.2 Isolation forest

Isolation Forest is an unsupervised decision-tree-based technique that was first designed for outlier detection in tabular data. It works by randomly dividing sub-samples of the data based on some attribute/feature/column. The notion is that the rarer the observation, the more probable it is that a random split on some feature will isolate outliers in one branch, and the fewer splits it will take to isolate (form a partition with just one point present) an outlier observation like this.

If there is an outlier in the data, and we pick a column at random in which the value for the outlier point is different from the rest of the observations, and then we pick an arbitrary threshold uniformly at random within the range of that column and divide all the points into two groups based on whether they are higher or lower than the randomly-chosen threshold for that column, then there is a higher chance that the outlier will be found.

Outliers are not often defined by having a single extreme value in a single column, therefore a successful outlier identification algorithm must consider the relationships between numerous variables and their combinations. One such method is to construct an "isolation tree" which consists of recursively repeating the randomized splitting procedure outlined above (that is, we divide the points into two groups, then repeat the process in each of the two groups that are obtained, and continue repeating it on the new groups until no further split is possible or until meeting some other criteria).

According to this scheme, the more common a point is, the more splits it will take to leave the point alone or in a smaller group compared to uncommon points - as such, the "isolation depth" (number of partitions that it takes to isolate a point, hence the algorithm's name) in the isolation trees can be thought of as a metric by which to measure the inlierness or outlierness of a point.

A single isolation tree has a lot of expected variability in the isolation depths that it will give to each observation; thus, for better results, an ensemble of many such trees - an "isolation forest" - may be used instead, with the final score obtained by averaging the results (the isolation depths) from many such trees.

There are other potential methods to improve the procedure's logic (for example, projecting an isolation depth after reaching a particular limit), and the resulting score can be standardized for ease of use, among other things.

When compared to other outlier/anomaly identification approaches like "local outlier factor" or "one-class support vector machines," isolation forests have the following advantages:

- It is resistant to the occurrence of outliers in training data.
- Multi-modal distributions are robust.
- Variable scales are unimportant.
- Much easier to install.

It is insensitive to the distance measure used (since it does not use one in the first place). Furthermore, because they output a standardized outlier metric for each point, such models can be used to generate additional features for regression or classification models, or as a surrogate for distribution density, which is not equally acceptable for all outlier detection approaches.

As stated previously, Isolation forest outlier detection is simply a collection of binary decision trees. Also, each tree in an Isolation Forest is known as an Isolation Tree (iTree). The approach commences with data training by constructing Isolation Trees.

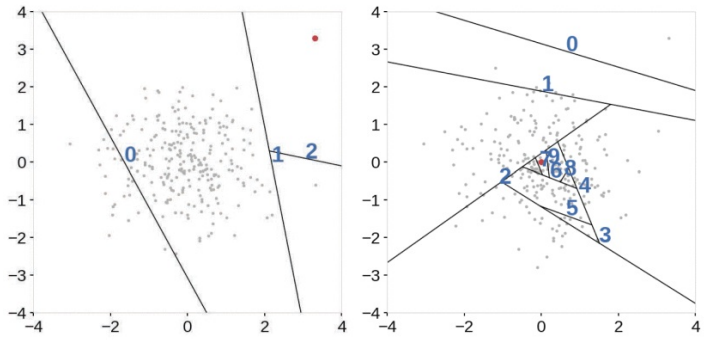
Let's examine the entire algorithm step-by-step:

- A random subsample of the data is selected and assigned to a binary tree when a dataset is provided.
- The tree's branching begins with the selection of a random feature (from the set of all N features). Thereafter, branching is performed based on a random threshold (any value in the range of minimum and maximum values of the selected feature).
- If a data point's value is less than the specified threshold, it is routed to the left branch; otherwise, it is sent to the right. Hence, a node divides into left and right branches.
- This technique is repeated recursively until each data point is completely separated or until the maximum depth is attained.
- The preceding procedures are performed to generate random binary trees.

Model training is complete after an ensemble of iTrees (Isolation Forest) is generated. A data point travels through all previously trained trees during scoring. Now, an 'anomaly score' is applied to each data point based on the tree depth required to reach that point. This score is an aggregation of each iTree's obtained depth. A -1 anomaly score is assigned to anomalies and a 1 anomaly score is allocated to normal points based on the contamination (percentage of data with anomalies) parameter provided. Figure 2.16 illustrates the realization of Isolation Forest around two different datasets.

2.3.3 Conclusion

This chapter took a glimpse at a large quantity of related work on software log mining tasks. It also provided a review of the data analysis and machine learning techniques used in this research. To conclude this chapter, it can be mentioned that there is already a large amount of literature in the field of log mining tasks that deals with similar topics such as fault prediction, anomaly detection,



(a) Anomaly (b) Nominal

Figure 2.16: Process of branching in the Isolation Forest [146]. The branching process for an anomalous data point is depicted in Figure (a). The branching occurs till the isolated point is reached. In this instance, three random cuts were sufficient to isolate the location. Figure (b) depicts the same process of branching for a nominal point. Since the point is close to the center of the data, multiple cuts are required to isolate it. In this instance, the depth limit of the tree was reached prior to the identification of the point.

and test log minimization. However, these approaches do not fulfill all the requirements that are specifically addressed in the Philae objectives.

Addressing the gap between the existing methods and publications and the requirements of Philae's case studies was the focus of this research. We must mention four main deficiencies in the published methods:

- *Application-specific solutions*: software artifacts differ in many aspects, from their architecture, the nature of the data that they process, their response time, and network activities to their log output information, as well as in the nature of their faults, the complexities of their causality, and how their effects are projected on the log files. And this is only if we want to name a few. Therefore, available approaches are either limited to a specific software environment or they make some basic and general assumptions about how software behaves. This fact limits the generalization of the existing methods in some particular cases. For instance, in the coming chapter, we will show why existing solutions cannot address *status monitoring* log files.

  On the Philae project, we dealt with two different case studies, which were augmented to three with another open-source software. We had different log mining tasks to accomplish on each case study. Therefore, we were seeking a methodology to consolidate all the log mining tasks from all case studies into a generalized approach where we could create a machine-learning model to address them all.

- *Discovering mutual and multi-causation effects*: To the best of our knowledge, none of the existing approaches are meant to find mutual effects of different events to trigger a bug. For instance, in Philae case studies, some anomalies were caused by a group of events in some specific order of appearance. Hence, one gap to fill was to learn about these complex causalities between events and failure or anomaly.

- *Having fewer empirical variables*: dealing with different scales and complexities of different software, we must tune proposed methods to be scaled appropriately with the software. For instance, in noisy log files, the anomaly detection threshold must be precisely tuned to avoid false alarms. One gap to fill from the related work was to have an approach with fewer empirical variables. This makes the proposed approach more generic, less error-prone, and easier to deploy.

- *Unsupervised learning*: On top of all the above-mentioned shortages to address, unsupervised learning gains great importance. Software logs could be huge and creating a learning data set out of them for supervised learning may require considerable time, and in some cases, it might be infeasible. Unsupervised learning creates a fast and easy model creation without minimal manual intervention or pre-processing for labeling the data on the log files.

In this regard, this dissertation advances the state of the art by providing log analysis and machine learning techniques to log mining tasks to reach Philae objectives.

Chapter 3

The Log mining Methodology

Contents

3.1 Introduction

Logs are generated by log statements in software source code. Developers insert log statements to expose and register information about the internal behavior of a software artifact in a human comprehensible fashion [147]. Log inspection is a general way that helps developers in some software maintenance activities such as testing, debugging, predicting, and diagnosis. The content and format of logs can vary from one software system to another.

The classical viewpoint on software testing assumes that for each given input entry, the software returns an output (or a log event) record which are distinct from the other input-output (or input-log-event) pairs. Accordingly, assigning "Pass" or "Fail" labels to the output logs is mostly feasible based on the input and the expected software functionality. These separated "input-output" or "input-log" pairs form a basis to test a software artifact or perform some post-processing steps on test-suites, like "regression testing" or "test-suite reduction" [10]. 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, much 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 [10]. 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 to test/log analysis, prediction, modeling or even reduction [63]. We called this situation *Software-Level Activity (SLA) logging* due to the fact that the output log is a trace of software activities and outputs. The format of log data in *SAL logging* can vary significantly among software systems. They typically capture events by recording the time when the event occurred, information about which user caused the event, and details about the software system's reactions.

In contrast to the SLA logging, for certain types of software, associating a fault situation to a specific input-output is not explicit. Instead, the internal faults drive the computer system into a period of anomalous behavior, which may end up in a system failure. Many of complex software systems experience similar situation. For instance, a network appliance, a cellphone, cloud infrastructures or a multi-user operating system may experience a period of anomaly that ends up in a system reboot. In such systems, there is a time epoch between a failure and the input that caused the failure. Knowing the period of anomaly and localizing its root cause input are in favor 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. In the above-mentioned condition, when gathering *SLA* logs and outputs is not feasible, a practical way to find anomalous behavior and their root cause input is system-Level *Monitoring logging*. In *Monitoring logging*, software testers sample the device's status or monitoring information (e.g: memory/CPU usage, number of processes, etc.) and then study this status information to find anomalous behavior. In *Monitoring logging*, we can record some different metrics. Some metrics include the host or container CPU usage, memory utilization, and storage capacity. These metrics give a broad understanding of the infrastructure's status and how well it suits the application's requirements. However, other types of metrics, such as the slowest and most time-consuming requests, provide a deeper understanding. The *Monitoring logging* has an intuitive property: The rates of input arrivals and status sampling can be different, and generally, the status information is sampled in relatively slower pace than the input arrivals. A gigabyte network appliance is such an example, in which, the rate of data arrival is thousands of

times faster than the possible status logging. In other words, the variations of the sampled values in status logging are significantly slower than the arrival rate of the inputs. Therefore, this approach has two serious challenges: the meaning of *error* is not directly linked to a specific input. Thus, we search for anomalous behavior instead of errors. But the second challenge is to relate inputs to the anomalies. In other words, a detected period of anomaly in the system spans over numerous inputs. Hence, finding an input or inputs that caused the anomalous behavior is challenging due to the slow pace of status sampling.

3.2 Top View of the Proposed Method

The general workflow of the proposed method is illustrated in Figure 3.1. It consists of three major steps: i.e., log analysis, model creation, and log mining tasks. In this section, we describe the general workflow of our proposed method. First, we elaborate on log analysis. Then, we describe model creation, and, finally; we introduce some major log mining tasks which we studied during this thesis, including failure prediction, root cause detection, test suit minimization and user behavior clustering.

3.2.1 Phase 1: Pre-processing on Software-Level Activity and Monitoring logging (System-Level Monitoring Logs)

The primary difficulties in working with logs, such as coping with inconsistent formats and extracting events from logs, is at the focus of several published methods for log analysis to understand user or system behavior, identify failure and its causes, locate abnormalities, ensure application security [15], and predict fault, log analysis attempts to extract data and information from logs to cluster hosts [148]. In this section, we go into greater depth about how our study's log analysis worked. We propose some preliminary steps to prepare log files to be processed by machine learning approaches. In this part, we distinguish between the log files with software activities and outputs (formerly called *SLA logs*) and those that only store the computer status information on time intervals (formerly called *Monitoring logs*).

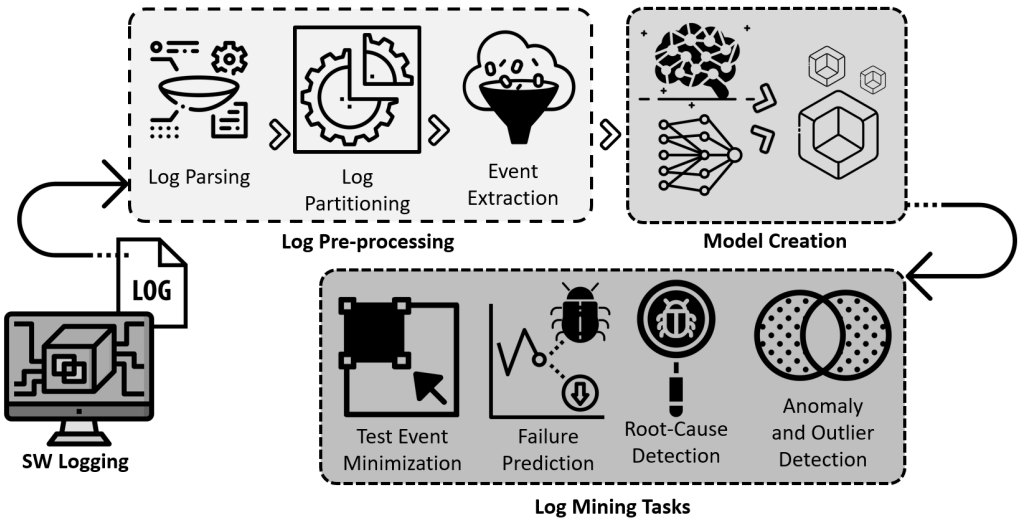


Figure 3.1: Top View Of the Proposed Method

3.2.1.1 Log Parsing

Log parsing is typically required for machine learning approaches. They require numerical vectors as input data. It is essential to parse logs and transform their events to vectors because logs are semi-structured text. The first crucial step is to extract log messages into structured data for further analysis because logs are inherently unstructured or semi-structured. The first and most important stage in enabling accurate analysis is log parsing, which converts a stream of structured events from free-text raw log messages [123]. The log message typically includes constants and variables. Constants are the fixed language that developers write to explain system events, such as "Fail" or "Pass". For each repetition of the event, they remain constant. Variables, on the other hand, are the values of program variables that include dynamic runtime information (i.e., parameters), which can change depending on the circumstances of an event [39]. A structured log message is produced by a log parser.

Since there are many different event templates due to the complexity of the software, log parsing is still a difficult operation. Additionally, the frequent updating of the logging statement is caused by the frequency of program changes. Several automatic log parsing techniques exist, including [149, 150, 151]. Both offline and online modes are supported by some of them [149, 151]. Offline log parsers need all the log messages in advance and parse the log messages according to rules. Online parsers, in contrast, parse the log messages as they come in. Some parsers preprocess the logs by deleting some variables or swapping them out with constants based on domain knowledge.

In the past, primary log messages were parsed using regular expressions to retrieve events. However, manually developing the rules to parse a log message takes a lot of time. Numerous research studies have suggested automated log parsing techniques, such as frequent pattern mining [96], iterative partitioning [15], and parsing trees [152], to reduce the manual labor involved in log parsing.

To show an example of how log parsing may be carried out, we refer to log parsing in a case study in our research. In the scanner case study, we employed test suits, a set of test cases designed to be used to test software programs. Since any action-input-output combination may result in a distinct behavior of the system, we decided to distinguish between comparable events that have different inputs and outcomes. This is generally the case for the majority of software artifacts, and one can parse logs by distinguishing all or a subset of "input", "action" and "output" (or occasionally "timestamp"). Each action-input-output was represented for this reason as a triplet vector with the notation $[a, p, o]$, where 'a' stands for the action. The input parameter is 'p', while the output parameter is 'o'. As a result, triplets can be used to encode all operations, inputs, and outputs. However, the idea can be used generically for any kind of software with any number of input and output parameters. In chapter 5, we go into great detail about it. In the Telecom case study in chapter 4, the log files contained a substantial record of inbound events spanning six months, as well as information on the device's status or monitoring. A lengthy list of input events with timestamps make up each test log. With samples obtained every 5 minutes, each monitoring record represents a full day of monitoring. 26 metrics are contained in each sample. chapter 4 expands on this in further detail. We explain how to connect these two sorts of logs in chapter 4. Additionally, we looked at the Train Ticket benchmark dataset. We describe it in chapter 4.

3.2.1.2 Log Partitioning

In complex software systems, the effects of different input events, coming from different sources, are interleaved and projected altogether on the output log. The purpose of *log partitioning*, is to divide our logs to extract independent sequences of events. This makes the inputs to the learn-

ing process less noisy and facilitates the machine learning's ability to distinguish among different behaviors of the event sources.

Log partitioning is an important step in creating learning materials based on the target of the machine learning problem. Here, we partition the input events into *input sequences* based on a specific *attribute*, around which, all the ML data mining will work and yield results. For instance, in a shopping application, if one partition logs based on their "customer ID", the ML would distinguish among users and we can assume final artifacts based on customer distinction, such as customer behavior clustering, customer behavior prediction, etc. Later, we can filter out similar customers and only focus on the customers with abnormal behavior based on the model created from the users' behaviors. Likewise, if we partition logs based on either "product ID", "days of a week" or "time" attributes, we would have different distinctions based on the chosen attribute.

In software-level activity logging, the timestamp and log identifier are often used to divide the log into sections. The timestamps in different formats can be easily extracted from raw logs in the log parsing phase (Section 3.2.1.1). It records the occurrence time of each log, which is a basic feature supported by many logging libraries. On the other hand, a log identifier is an identifier that indicates a series of related system actions or message transfers. For example, shopping software uses client ID to identify customers who perform various actions (such as scanning, deleting, and paying) during a particular session. Common log identifiers such as user ID, task/session/job ID, and so on can be extracted using log parsing. Log identifiers are a more explicit and clear indication for partitioning logs than timestamps. In our study, we divided logs using timestamp and customer ID in the Telecom and scanner case studies, respectively.

In monitoring logging, input events do not have a direct and exclusive link to the monitoring logs. Instead, the accumulated effect of a large number of input events is superposed and projected onto the status information. Therefore, correlating an individual input event to a status change in the monitoring log is not a straight-forward task. In other words, a system status record does not belong exclusively to a specific user, session, or source, and as a consequence, one cannot partition monitoring logs by relating them to different sources. To address this issue, we proposed to first pre-process the monitoring log to find anomalous time periods and their start and stop timestamps.

As a contribution of this research, to partition monitoring log files, we developed the idea of *Bug-Zones* and employed it in the Telecom and Train Ticket case studies. A Bug-Zone is a period of time during which a software system shows anomalous behavior. As described at the beginning of this section, monitoring logs or status information don't explicitly convey fail and pass conditions (unlike software-level activity logging). As a result, the Bug-Zone concept was a trial run for developing a meaning for a fault in monitoring logs where the normal continuation of the software system is perturbed for a period of time due to an abnormal internal condition and its effect appears on the status information. We developed a log-partitioning tool based on Bug-Zones (Bug-Zone finder). The Bug-Zones Finder includes the following steps: anomaly detection, outlier counter sliding window, standardization, outlier density curve generation, and Bug-Zone extraction. The first step in finding Bug-Zones is to use outlier detection functions to preprocess the monitoring data. It is possible to apply various techniques of outlier detection and then vote among them. The outlier counter sliding window simply counts the number of detected outliers inside a specific sliding window. The sliding window creates higher values as the number of outliers in a specific period of time increases. Then, a standardization pass removes the mean value of the sliding window output and modifies its standard deviation to one. The output is what we call the *Outlier Density Curve (ODC)* in chapter 4. Bug-Zones occur when the outlier density curve exceeds a threshold line. The general workflow of log partitioning is illustrated in Figure 3.2, which mainly consists of three main calculations: outlier detection, outlier counter sliding window, and

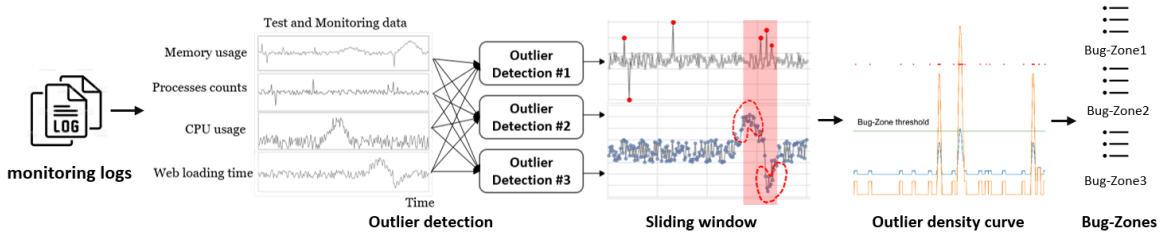


Figure 3.2: Example of log partitioning in Telecom case study

outlier density curve with standardization. We go into great detail about it in chapter 4.

The Bug-Zones, extracted from the monitoring log in the previous step, enable us to extract their potential causes among the input events. In this case, we always assume that the Bug-Zone's cause or causes occur prior to the Bug-Zone's start, within a limited delay before its effect appearance on the status information. Hence, the next step in monitoring log partitioning is to extract input events that occurred before the Bug-Zone (Pre-Bug-Zone). The input extraction time range is determined by the system developers' observations made on the outlier density curve, taking into account how long the root cause may have occurred prior to the Bug-Zone. The details will be covered later in chapter 4.

3.2.2 Phase 2: Machine-Learning Model Creation

The model creation step works on the pre-processed logs to create model artifacts ready for data mining usage, as presented in Figure 3.3. The model creation has four steps:

- Input event representation
- Input sequence representation
- Feature selection
- Creating Universal Clusters (UC)

Each step will be covered in the following subsections.

3.2.2.1 Input Event Representation

The first proposed step to create an ML model from the input events is to use a word embedding approach to create a vectorized representation of the input events. Consider a set of log files related to a software artifact, each of which, contains a different number of input events. As we treat input event sequences as sentences of words in a natural language, we analyze these log files using sequential tools developed for Natural Language Processing(NLP). Therefore, the terms *word* and *input event* have the same meaning and are used interchangeably.

Word2Vec [136], as the main NLP tool in this research, is one of the widely used word embedding approaches. Word embedding is a group of word representation methods that extracts the similarity or closeness of the words in a series of sentences and represents them in a numerical form. Word2Vec creates a mapping from a text corpus's word set (so-called *vocabulary*) to an Euclidean space. In this thesis, every distinctive word (or input event in our application) in the vocabulary is assigned to a corresponding vector in the Word2Vec space. The distance between the words indicates their semantic relationship. Hence, two words with a close Euclidean distance in the Word2Vec space must have a close meaning. Therefore, the first step in the model creation phase is to extract the semantics of the input events.

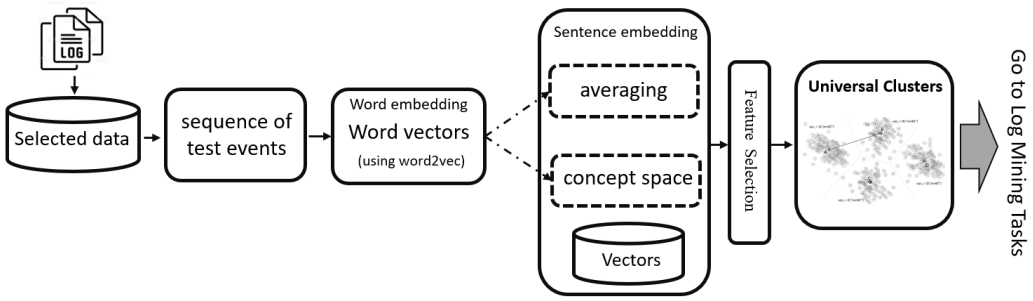


Figure 3.3: Model creation overview

For instance, in the scanner case study, we partition the sequences of events (a session) that pertain to a particular client. Then, to acquire all information of the client's activities, each session is shown as a sequence of triplet vectors ([action, input parameter, and output]). As a result, we have some triplets for each session, and we treat them like words in NLP. By considering input events as sequences, we were able to discover sessions that were semantically related via Word2Vec. It should be noted that we store each triplet as a string (e.g: '[delete, barcode, 1]') and we treat them like words in natural language processing. For instance, in the following paragraph, it is a 15-element vector generated by the Word2Vec method that represents the triplet [scan, barcode, 0].

```
[scan, barcode, 0] :
[ 1.2445878, 1.613417, -0.1642392, 3.0873055, -0.355896 , 1.0599929, -0.49392796, 1.0838877, -1.1861929,
-0.2639794, -0.09810112, -0.9824149, 0.881457, -3.6238787, -1.1903458 ]
```

It must be noted that we are dealing with high-dimensional vectors and to reach a visual depiction of the created models, we used the t-distributed stochastic neighbor embedding (t-SNE) method [153] to show the syntactic and semantic relations of the words in two-dimensional space. Figure 3.4 shows the triplets' Word2Vec vectors from the shortest log file of the scanner case study. In this log, we have 61 sessions and 15 distinct triplets, hence the Word2Vec dimension for each vector is 15. These 2D visual depictions are useful to understand how ML is seeing the data on each step.

The input event representation step is identical for SLA and monitoring logging. The only main difference is how the input event sequences are generated for each category: The sequences of events in SLA logging have been created by log partitioning based on timestamps and log identifiers (*e.g.*:userID, sessionID etc.), while the event sequences in monitoring logs have been created by Pre-Bug-Zone test extraction, as was described in the previous subsection. These distinctions will be discussed in the chapter 4 and chapter 5 Scanette (SLA logging case) and Telecom case studies (monitoring logging).

Furthermore, in the Telecom case study, a model is built using input events extracted from test logs. We extracted input events that occurred before the anomalous period of time from the monitoring logs in the previous phase. Pre-Bug-Zone sequences are what we call them. We then extracted random time intervals from time ranges that were not in the Pre-Bug-Zone. Each Random-Zone or Pre-Bug-Zone input array is considered a sequence. Each input event in that array is similarly treated as a one-hot-coding vector.

In order to implement our Word2Vec model, we utilize Gensim [154], to produce a set of word embedding by the dimensionality D and the context window size W. We go over these examples in detail in chapter 4 and chapter 5.

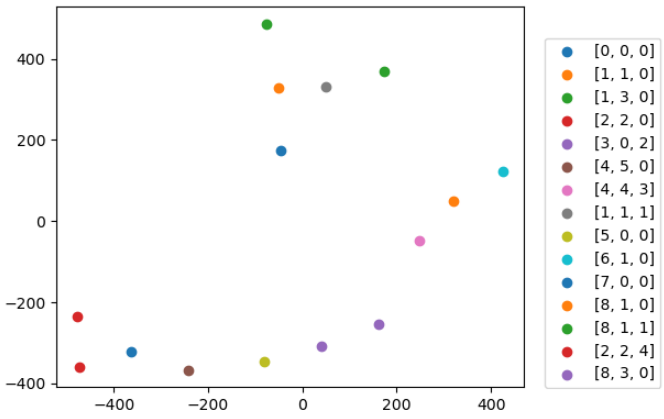


Figure 3.4: 2D results of the Word2Vec vectors for the scanner case study.

3.2.2.2 Sequence Representation

After the first step, each word has a numerical vector representation and the distance between two vectors determines how their two corresponding words are semantically close. So far, each sentence (sequence of events) is a sequence of vectors. The next step is to make a single vector representation for each sentence.

A sequence representation vector is a superposition of all words in that sentence. This step enables us to measure the similarity of the sentences. There are different methods to create a single vector from a sequence of vectors [138, 141, 155, 156]. As we illustrate in Figure 3.3, we employ two different methods in our model for sentence embedding; *sentence averaging* and *concept-space creation*.

Sentence Averaging: In fact, a common method to achieve sentence representations is to average the word representation vectors. This is a basic and common method for creating distributed sentence embeddings that do not take word order into account.

Concept-Space Creation: In addition to comparing the averaging method's results, we propose the concept-space method, which is based on a similar idea expressed in [157] and uses vectors to represent each sequence. The steps required for creating concept space from test events are depicted in Figure 3.5. To achieve concept space vectors, first, we clustered input events (or words) based on their semantics into groups of similar events. As a result of using a clustering method (typically K-means), we get $Concept = [con_1, \ldots, con_n]$. Then we referred to each group as a *concept*. In fact, each vector characteristic corresponds to the proportion of events from clusters that appear in the sequence. For example, in Figure 3.6, the input events of the Telecom case study are plotted in part (a). After using the K-means approach, the identified concepts are displayed in part (b).

After developing the concepts, it is feasible to determine the conceptual presentation of an event sequence by observing its events and the concepts to which they belong. For example, we have a sequence of tests like $S_1 = [t_1, t_2, t_3, t_4, t_5, t_6, t_7]$. After finding how many input events in this sequence belong to each concept, we can have one vector that represents this sequence. For instance, [0,2,0,3,2,0] means that we have two tests from concepts 2 and 5, three tests from concept 4, and no tests from concepts 1, 3, and 6.

The efficiency of simple averaging and concept space representations will be compared in an example in the Train Ticket Benchmark subsection in chapter 4.

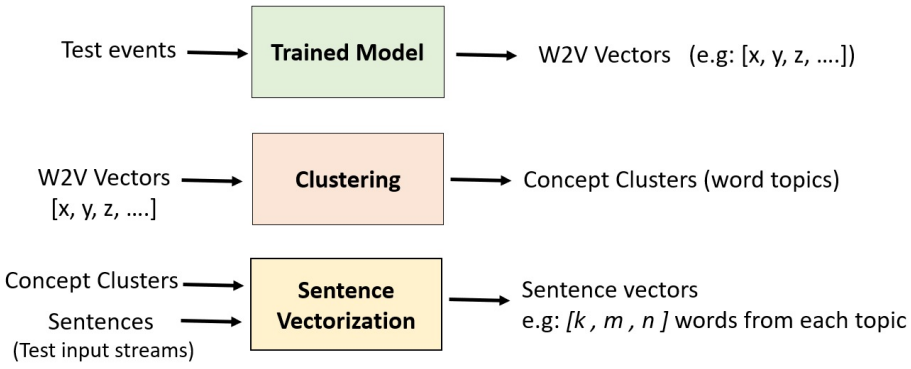


Figure 3.5: Sentence representation by concept space creation

3.2.2.3 Feature Selection: extracting important dimensions

Some datasets with a large number of features (or dimensionality) and few samples tend to be particularly prone to overfitting. The use of an overfitted model may lead to errors during the research, which may then lead to more errors. Noisy features can also amplify the difficulty. Noisy data has a tendency to affect machine learning algorithms. As the dimensionality increases, the computational cost also increases, usually exponentially [158]. To overcome this problem, it is necessary to reduce the number of features that are being considered. Two approaches are usually used to reduce the number of features: 1- feature subset selection and 2- feature extraction.

Feature subset selection: It is one way to remove redundant and irrelevant features. It does not modify the original data representation. One objective for feature subset selection methods is to avoid overfitting the data in order to make further analysis possible [158]. There are three distinct feature selection algorithms: the *filters* that extract features from data without any learning involved. The *wrappers* that utilize learning methods to find significant features. And the *embedded techniques* which combine the feature selection step and the classifier construction.

Feature extraction: In contrast, in order to reduce the dimensionality of the selected features, feature extraction creates new variables through the combination of others. Feature extraction serves two purposes: separating useful information from irrelevant data and decreasing the number of classification procedures by reducing the dimension. Its advantage is that new features can be compressed more efficiently; a disadvantage of this is that the initial feature set has a specific meaning, and the new features may lose their meaning [159].

A well-known feature selection technique is random forests. It is a collection of classifiers. Feature selection using Random forest falls under the category of embedded techniques. Embedded methods are a combination of filter and wrapper methods. The forest with the smallest amount of features and the lowest error is selected to be the feature subset. In some studies, it was demonstrated that the RF approach has high precision among all categories and is the best classifier [160], [161]. In [162], shows that among of the machine learning techniques, Random Forests(RF) have been an excellent tool to learn feature representations.

Another example of feature extraction is PCA (Principal Component Analysis) method. PCA is a statistical analysis technique that, from the perspective of feature validity, transforms numerous feature indicators into a small number of extensive indications. Even though PCA is excellent for the vast majority of feature extraction, it is a linear model, which may be unsuitable for some datasets. Another technique for dimensionality reduction is t-Distributed Stochastic Neighbor Embedding (t-SNE) [139]. For each point, it determines which other points are its "neighbors" and attempts to ensure that each point has the same number of neighbors. Then, it attempts to embed the points so that each has the same number of neighbors.

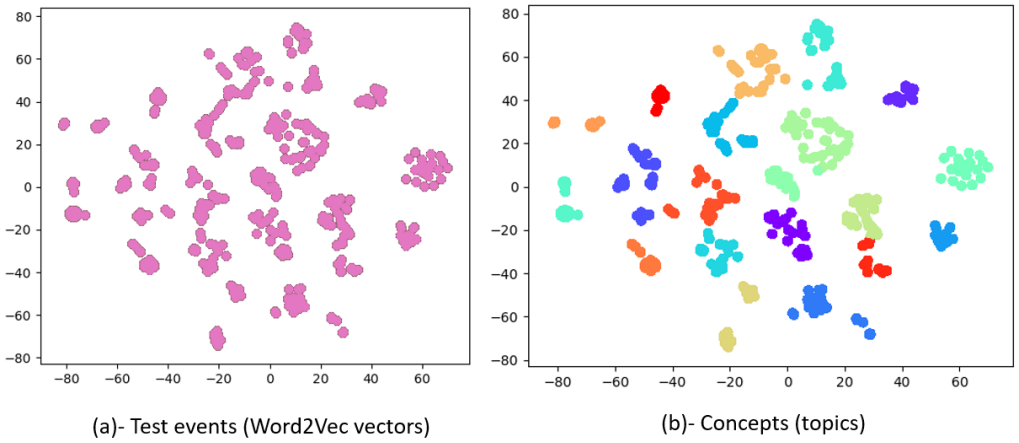


Figure 3.6: The concepts (input events) for the Telecom case study.

The choice of feature selection or extraction depends on the data set's characteristics, including noise, amount of available data, and variety of events. In some cases, feature selection removes noisy dimensions, which in turn leads to better accuracy and speeds up processing by having lower-dimensional data. On the other hand, in some other cases, it might be not effective or even remove some features that convey important information.

3.2.2.4 Universal Clusters Construction

The final step of the model creation is to construct the Universal Clusters (UC) by clustering the sentence vectors. These clusters are essential to the verdict on sentences' semantics and their behavior for log mining tasks such as failure prediction. In fact, if the model creation step effectively succeeds in extracting meaningful information from the input event sequences, then we would expect to see that a particular cluster covers a group of input event sequences that all show some similarities in their semantics, for instance, in terms of input events and their order of appearance. We can also state that the UCs project all possible topics (concepts) in the input event sentences. Each input sequence should belong to one of these UCs. The distance between an input sentence and UCs is used for prediction or root cause detection tasks. These will be covered in the following section.

The way to use the UCs depends on the log mining task and will be covered shortly after this subsection. For example, depending on whether the majority of the test sequences in that UC result in a fail condition, the UC may be labeled as *Pass* or *Fail*. Likewise, the center of a UC may gain importance if we want to know to which cluster a new test sequence belongs and hence shows similar semantics or effects as the sequences belonging to that UC. The UC center is simply an element-wise averaging of the cluster members, and the label is the same as the label of the majority of the cluster members.

As Figure 3.7 shows, a clustering algorithm must be employed to create UCs from all sequences. It must return a set of clusters $UC = \{uc_1, \ldots, uc_U\}$, each with a corresponding center calculated by averaging the members of the cluster, and possibly a label that indicates if the majority of the cluster members trigger the failure. The label is generally used for prediction tasks. Therefore, we would have a set of UCs and each of which is designated by its center and its label:

$$UC = \{uc_1, \ldots, uc_U\} \tag{3.1}$$

$$uc_i = (center_i, label_i) \tag{3.2}$$

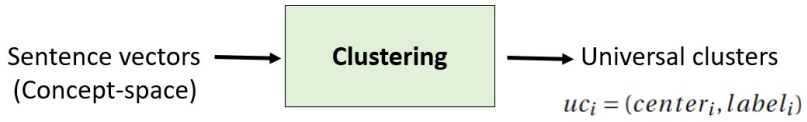


Figure 3.7: Universal Clusters Construction.

3.2.3 Phase 3: Log Mining Tasks

Log mining employs statistics, data mining, and machine learning techniques for automatically exploring and analyzing a large volume of log data to glean meaningful patterns and informative trends [39]. The generated structures and data could help in the management, monitoring, and troubleshooting of software systems. Finding meaningful failure logs manually is similar to looking for a needle in the desert. In addition, software and hardware faults may cause failures due to the complexity of advanced software systems. Therefore, advanced strategies for implementing automatic log mining are in high demand. This part introduces four important log mining problems, including root-cause detection (section 3.2.3.1), online failure prediction (section 3.2.3.2), test log minimization (section 3.2.3.3), and user behavior clustering (3.2.3.4). For each of them, we explain how our proposed method can help to achieve these objectives.

3.2.3.1 Root-cause Detection

In general, the goal of root cause analysis methods is to identify the precise source of an issue so that corrective measures can be taken to prevent its repeat. After detecting an anomaly or identifying a performance issue, the analyst determines the underlying cause by examining and interpreting a vast volume of log information. System administrators can obtain significant information for root cause analysis by correctly classifying and correlating log events. In this study, the objective of RCA is to identify the relationship between incoming events (such as inputs, network requests, new connections, new user logins, etc.) and any anomalous software behavior. Further research into detecting root-cause in software source code goes beyond the goal of Philae case studies and, as a result, this research.

Universal Clusters are meant to cluster the *bug-triggering* test sequences in the same clusters. As a result, they can point software developers to tests that cause a test sequence to fall into a bug-triggering UC. The software developer can later focus only on a limited number of bug-triggering tests and ignore the others. To extract this root-cause information from UCs, we proposed two different methods, 1) Subtracting UC centers and 2) Mining the model created by the UC clustering algorithm. Here, we give a general view of the two methods:

1) *Subtracting UC centers:* The UC centers, as averages of all sequences in that cluster, can reveal the features that are distinguishable between a bug-triggering cluster and a *safe* cluster. For example, in an abstract illustration in Figure 3.8 which is drawn in two dimensions, there are four UCs, each contains several test sentences extracted from *pass* (non-Bugzone/Randome-Zones) and *fail* (Pre-Bug-Zone) periods (Pre-Bug-Zone and Random-Zone are two different extracted tests sequences in Telecom case study). In each cluster, one of these two categories is in the majority. The cluster label is the same as the label of the majority of members. A and B represent the centers of two UCs. By subtracting the two centers A and B, one labeled as *pass* and one labeled as *fail*, one obtains a vector such as $[diff_1, \ldots, diff_C]$. The index of the largest value (e.g: f_{imax}= MAX $[diff_1, \ldots, diff_C]$.) indicates the concept with the highest contribution to the bug triggering. Therefore, the members of $Concept_{imax}$ are the potential root cause of the anomalies, since their presence differentiates members of A from members of B. It should be recalled that the members of $Concept_{imax}$ are words that are equivalent to input event types that directly

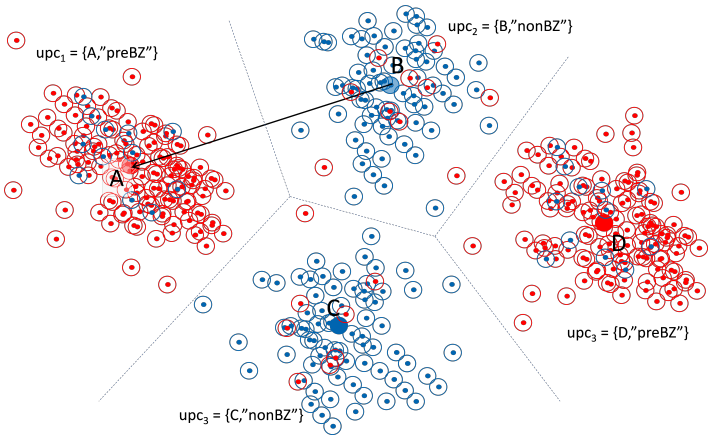


Figure 3.8: Universal Clusters and finding event root-causes

contribute to creating anomalies. Therefore, the observed error or anomaly is susceptible to being triggered by the events of the $Concept_{imax}$.

2) *Mining the model created by the UC clustering algorithm:* The second method for determining the root cause is to examine the model created during the UC creation step. During the UCs creation, a clustering algorithm distinguishes between fail and pass sentences and clusters them based on their concept-space vectors. Usually, the clustering algorithm creates an internal (probabilistic) model from the inputs. From that model, we can extract the features (concepts) importance or their discriminative power to tell clusters apart [163, 164]. The most important concepts and their input events are suspects in the abnormal behavior.

logistic regression is another method for determining the significance of the features. Logistic regression is used to analyze data and the relationship between a dependent variable and one or more independent variables. Independent variables may be nominal, ordinal, or interval in type. We can train it using Universal Clusters data. It fits a logistic function curve based on Universal Clusters' features. The goal of logistic regression is to identify coefficients that appropriately fit the data and minimize error. Since the logistic function returns a probability, we may use it to order possibilities from least probable to most likely. We use logistic regression in the Telecom case study for the same purpose. This will be covered in chapter 4.

3.2.3.2 Online Failure Prediction

Online failure prediction provides system administrators with advance warning of imminent abnormal situations and possible system failures. To create an online predictor, we need only train a classifier using a set of sentences labeled with different categories. The classifier learns the sequence classes that are more likely to belong to the first set and distinguishes them from the other sequence types. However, we suggest a second method that employs the created UCs centers as indicators to identify if a series of events may lead to a failure. Assume that the most recent events set that just arrived to the system is denoted by $LastInputs = \{I_1, \ldots, I_{3\tau}\}$. In the prediction phase, the conceptual vectorized version of the most recent input events can be calculated from the created model: $LastInput^{Concept}$=$[con_{i1}, \ldots, con_{iC}]$. It must be noted that the output vector has a dimension of C, regardless of the number of input events. The closest UC to this vector determines the prediction verdict. For instance, we imagine that the smallest cosine distance is between $LastInput^{Concept}$ and UC with a fail label. Therefore, the predictor predicts a failure 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 value in $LastInput^{Concept}$ must be decremented, and another one must be increased. The cosine

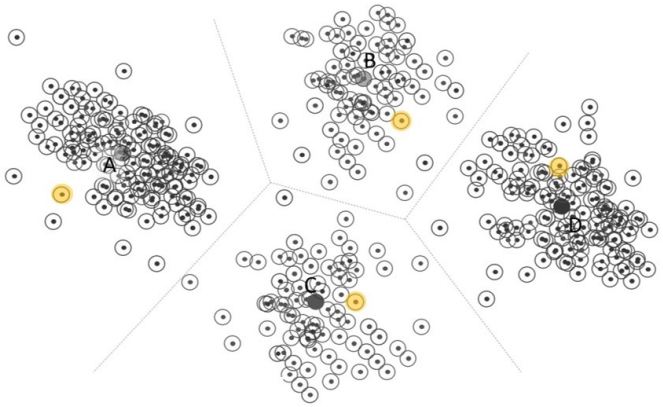


Figure 3.9: Sessions selection from Universal Clusters

distance must be calculated again to find the closest UC.

Like any other online predictor, the proposed method is a subject of false positive, false negative, and computational complexity estimations that will be covered in the case studies.

3.2.3.3 Test Suite Minimization (Log Reduction)

A common issue in software testing research is this: Given a piece of software S and a corresponding test suite T, how can we quickly determine whether S passes on T, or if not, which test cases failed? The first obvious solution is to run the entire T and observe the S's output, but this is not considered an *efficient* solution since software input events are generally a large number of test sequences. The purpose of this log mining task is to use the created model from the previous section to reduce or minimize a test suite to a considerably lower number of input events while the reduced set still has the same effect as the whole test suite T, or, in other words, they trigger the same fault as the whole T.

Here, we assume that the test suite T is composed of several sequences of input events. Referring to the previous section, the UCs created from the test sequences can be used to minimize T. As Figure 3.9 shows, a way is to simply take one representative test sequence from each cluster and run only those test sequences. In this case, we expect to see that the representative of each UC has the same effect as all UC members combined.

Here, we observe a compromise between the level of test minimization and the coverage of the whole T's effect. In other words, the more one reduces the number of input events, the more likely it will be that the minimized set has a less fault-triggering effect of the test suite T. One way to trade between the minimization level or fault-triggering effect is to create less or more UCs as described in subsubsection 3.2.2.4. The higher number of UCs means more fine-grained distinction between the semantics of the test sequences and also more representative test sequences, which, in turn, means a higher probability to trigger more faults.

The Scanette case study in chapter 5 is an example of test suite minimization. The effectiveness of the proposed method for this log mining task will be covered in that section.

3.2.3.4 User Behavior Clustering

User Behavior Mining (UBM), as was introduced in Section 2.2.3.1 is used in process mining (business model extraction, user behavior analysis, etc.) in complex software systems. Identifying higher-level actions from lower-level interaction logs can be challenging. This is due to the fact that it can be difficult to put together and then describe activity at a lower level. Preprocessing a log using trace clustering, which divides it into smaller groups of related traces, is one technique to deal with this issue. The ideal outcome of applying process discovery to those more coherent

sub-logs is a series of simpler, easier to understand process models that together give a better perspective of the entire log [165].

Here, we can consider two tasks for the extracted UCs in phase with UBM: 1) Users' behavior clustering, 2) User behavior prediction. Before briefing each task, we must note that we assume that each test sequence is a track of a particular user's actions, which we will refer to as a *session*. A session in an online shopping software, for example, could be a series of actions beginning with log-in, product search, adding to the basket, and payment. With this assumption, the UCs are created based on the semantics of user actions, and each member of a UC is a user session.

1) *User behavior clustering*: Universal Clusters intuitively divide users' sessions into several categories. The population of the UCs might give an insight into answering further statistical questions, such as UCs might give an insight into answering further statistical questions, such as "similarity of user behaviors", "diversity of their actions", etc. The distribution of a particular user's session among UCs might reveal information on his or her preferences. Finally, we may find overused and unused services and resources to remove system bottlenecks and optimize the system's performance.

2) *User behavior prediction*: Because we discovered user clusters that were instructive during the analysis of log files in our case studies, we decided to include UBM as one of the log mining objectives in this study. Due to the current surge in interest in ML-based UBM, we choose to look into this task in our log analysis.

3.3 Experimentation on Case Studies

Given this description of our proposed method, we aim to shed light on a number of empirical questions: (1): How effective is the proposed approach? (2): What is the complexity of the proposed method? To this end, we present an evaluation of our model over three different case studies. We present them in detail in chapter 4 and chapter 5.

3.4 Conclusion

In this chapter, we presented a method that aims to create an ML model from software logs by treating the input events as words in a word-encoding NLP scheme. The created model is evolved by extracting semantics and concepts from the vocabulary (input events) to establish a conceptual vector representation of each test sequence. Then, by clustering the test sequence vectors, universal clusters are formed to be the basis for some important log mining tasks, including root-case detection, fault prediction, test suite minimization, and user behavior clustering. The proposed method can also be adopted for status monitoring testing cases where the system status is monitored and processed to reveal abnormal behavior. In the next chapters, we will cover the log mining tasks in three case studies.

Chapter 4

Failure Prediction & Root Cause event Detection: Orange Livebox - A Telecom case study

Contents

The contribution of this chapter is a compilation of works published in the 22nd International Conference on Software Quality, Reliability, and Security- QRS (2022, China) and the 6th International Workshop on Software Faults- IWSF and SHIFT (2022, USA) [11, 12], in which we defined *Bug-Zone Finder* as an anomaly indicator tool. In the second work, we expand the first one 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 the imminent possibility of a system failure.

The chapter is included because it turns the theory of the proposed method (in chapter 3) into a practical solution for real-world industrial case studies. There, we demonstrate how the proposed method may be shaped to respond to the special nature of various software that deviates from conventional assumptions. This chapter will examine difficulties such as status monitoring (introduced in chapter 3), huge log files, non-specified *bug* effect, delayed failure (from the root cause event), and unsupervised fault detection and prediction.

4.1 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" [20] for such measurements. In many systems, all such events are stored in software logs, thus enabling post-production analysis. In this chapter, we are studying 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 behavior 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 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 behavior and finally terminates in a system failure. Complex computer systems, such as cell phones, network appliances, and distributed operating systems, are prone to such behavior, 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 behavior, 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 mature and operational 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 behavior. The analysis of monitoring logs must be an automated job due to the long sequences of data and rare anomalous periods [98]. In this sense, unsupervised machine learning can be deployed to achieve automated anomaly detection and online system failure prediction.

In this chapter, we apply the proposed method 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 behavior. 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 for 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 [13]. 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, this chapter will often refer to the implications of the approach on test logs. The results are presented in this chapter. Based on the work carried out for this study, a tool is published on GitHub [1] repository issued by the ANR PHILAE project.

The rest of this chapter is organized as follows. Section 4.2 describes the abstraction of the problem we are addressing in this chapter. In Section 4.3, we explain how we use our proposed method in detail. Section 4.4 explains our implementation and empirical results of our case studies. We discuss threats to validity in Section 4.5. Finally, we conclude this chapter in Section 4.6.

4.2 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 requests, 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 system failures at some points system failures. The failures could also be observed on the monitoring log. Figure 4.1 illustrates such a system.

Input events, which we call input events in our application context, are denoted by I=[I_1, ...,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 records events along with a timestamp. Therefore, an input event I_i is a couple made up of an *event type* which is a member of all possible input events, and a *timestamp* that records when the input 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 the

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

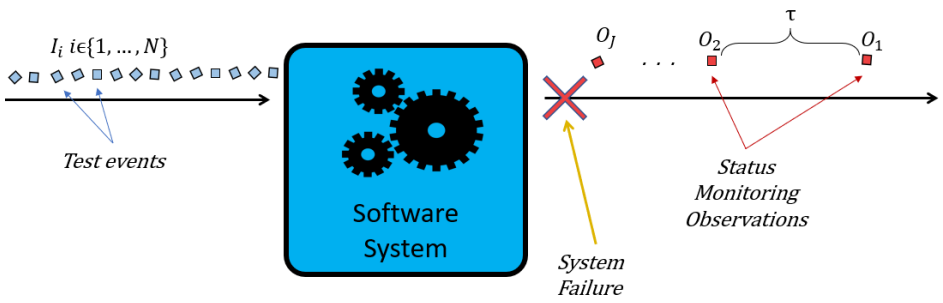


Figure 4.1: A software system with input and monitoring events

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 (Figure4.1).

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. 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 thesis 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. The authors of [17] 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 [166]. Furthermore, in the majority of approaches, the definition of fault is apparent [98] [19], 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 useful in our case.

There is some work on the automation of the log file analysis to identify failures or detect root causes, such as [17].In addition, there are also some efforts to work on predicting and localizing specific types of failures [16] [167]. For example, PatternMatcher proposed in [16] filters out normal metrics and unimportant anomaly patterns by using anomaly pattern classification and then ranks root-cause metrics. In [168] authors propose a method to predict disk failures before they cause more severe damage to the cloud system. they study both smart and system-level signals. Reference [81] presents a root-cause metric localization approach by incorporating log anomaly detection and correlation analysis with data augmentation.
Among limited published research on status monitoring logs, authors of [20] 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.

In this chapter, we present one of the few works that exploit system status monitoring obser-

vation 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 study 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. In this chapter, we also present the Train Ticket benchmark results for our proposed method. The implementation of the proposed method can be applied to many similar cases, either in testing or production.

4.3 Applying The Proposed Method On Case Studies

To elaborate more on the above-mentioned problems, we assume that a software system receives a chain of input test events. Examples of test events could be network packets, database queries, or API calls. The goal of our study in this chapter 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.

We apply the proposed method to two different case studies, Telecom and Train Ticket case studies. As Figure 3.1 illustrates, the proposed method consists of three major steps, mainly Log Pre-Processing, Model Creation and Log mining tasks. In this section, we provide an extensive description of each step.

4.3.1 Log Pre-Processing

A summary of how we implement the proposed method on case studies is shown in Figure 4.2. In the first and second part of the Figure (top of the Figure)

4.3.1.1 *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. *Bug-Zones Finder* contains these steps: Anomaly Detection, Sliding Window, Standardization and Generating Outlier Density Curve, and finally, *Bug-Zone* Extraction. They are discussed in detail below:

- *Anomaly Detection*

  To find these periods, the first step is to deploy outlier detection functions to preprocess the telemetry 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:

  $$A_q = OD_q(M) \tag{4.1}$$

  In (4.1), M= $(O_1,\ldots,O_J)$ 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= $[a_1,\ldots,a_J]$. 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.

- *Sliding Window*

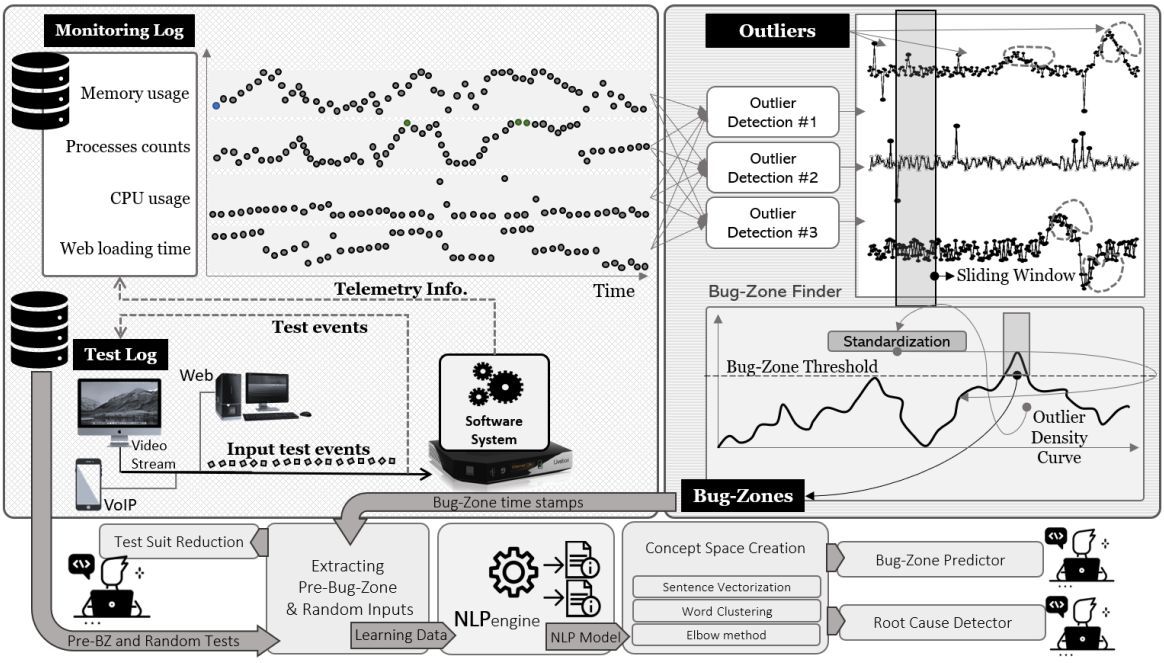


Figure 4.2: An overview of applying the proposed method to status monitoring and its associated log mining tasks

As shown in Figure 4.2, 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 4.3) :

$$A_{ac}[j] = \sum_{\forall A_q} \sum_{k=j-(W/2)+1}^{j+(W/2)} A_q[k] \tag{4.2}$$

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

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 4.2, we assumed that we had 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.

- *Standardization and Generating Outlier Density Curve*

  The properties of the output of the sliding window, A_{ac}, depend on several factors: the number of recorded monitoring features, the 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})

- 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 Figure 4.2). 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 $BZ \rightarrow T_E$.

4.3.2 Model Creation

The Model Creation phase has four steps:

- Input Event Extraction
- Model Construction
- Sequence Representation By Concept Space Creation
- Creating Universal Prediction Clusters

Each step will be discussed in the following subsections.

4.3.2.1 Input Event Extraction

At this step, one needs to extract input events in a time range before the *Bug-Zone* (Pre-Bug-Zone). Figure 4.4 depicts a single *Bug-Zone* period and the T second test input events that occurred before it. 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 input 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 4.1). 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*):

$$PreBZ = \{PreBZ_1, \ldots, PreBZ_Z\} \tag{4.3}$$

$$PreBZ_z = [I_{z1}, \ldots, I_{zP}] \tag{4.4}$$

$$Rand = \{RND_1, \ldots, RND_Z\} \tag{4.5}$$

$$RND_z = [I_{z1}, \ldots, I_{zR}] \tag{4.6}$$

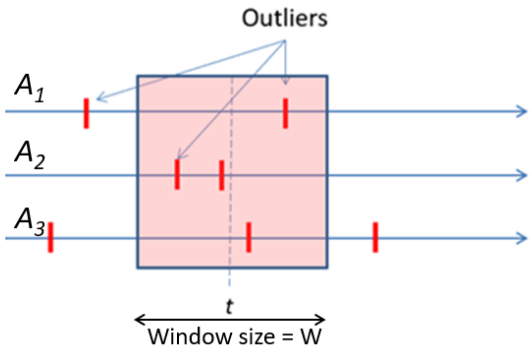


Figure 4.3: A sliding windows over anomaly detection arrays

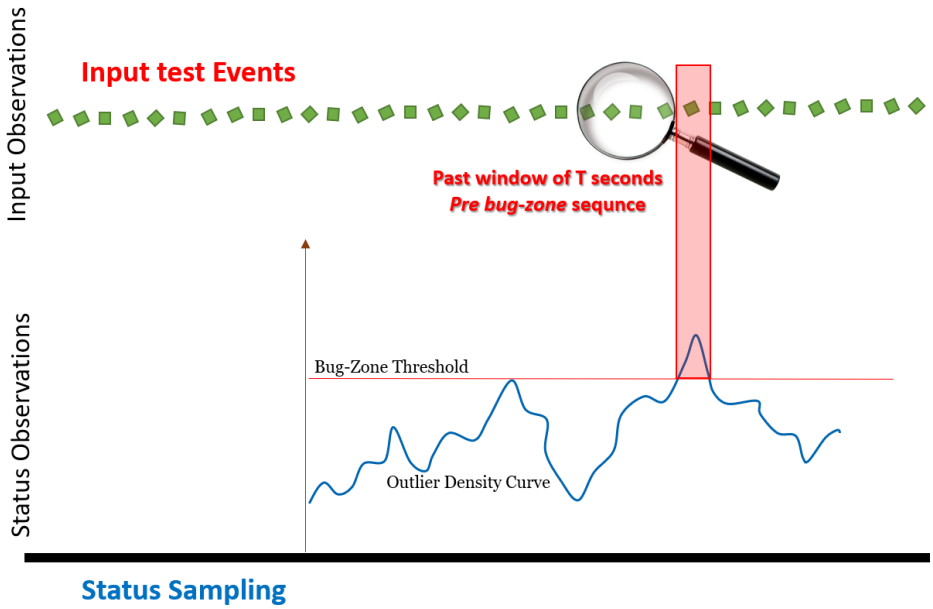


Figure 4.4: Pre-Bug-Zone extraction

In (4.4) and (4.6), 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.

4.3.2.2 Model Construction

At this stage, the extracted Pre-Bug-Zone input events are used to construct a model. Each Random-Zone or Pre-Bug-Zone input array is treated as a sentence of input events (words). Likewise, each input event in that array is treated as a one-hot-coding vector. We employed a contextual sequence model proposed by [136] to learn the representation of each input event. The model maps then each type of input events into a vector. The array size is $|\phi|$, in which, ϕ is a set of all possible input event types, called *vocabulary*. Likewise, the dimension of each vector in the array is $|\phi|$.

$$NLPModel = NLPEngine(PBZ, Rand) \tag{4.7}$$

$$NLPModel = \{V_1, \ldots, V_{|\phi|}\} \tag{4.8}$$

$$V_i = [f_1, \ldots, f_{|\phi|}], f_j \in \mathbb{R} \tag{4.9}$$

The generated $NLPModel$ model is comprised of $|\phi|$ vectors (*e.g*: V_i), each of which represents a word in the vocabulary, and in our case, a word is an input event type. Each vector V_i, itself, is comprised of $|\phi|$ real numbers (*e.g*: f_j). 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 the "sentence" term for an array of vectors representing an array of input events (e.g. a Pre-Bug-Zone input events).

4.3.2.3 Sequence Representation By Concept Space Creation

The created model gives vectors that represent the input 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 [10]. In contrast, our model creates a concept space from the input events by clustering them into groups of similar events and referring to each group as a concept based on a similar idea expressed in [157]. 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:

$$PreBZ_z^{Concept} = [con_{z1}, \ldots, con_{zc}] \tag{4.10}$$

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 $Rand_z^{Concept}$.

4.3.2.4 Creating Universal Clusters

The final step of the learning phase is constructing the Universal Clusters(UCs) from the sentences. These clusters are essential to differentiate between 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 UCs 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 clusters from all $PreBZ^{Concept}$ and $Rand^{Concept}$ sentences. It must return a set of clusters $UC = \{uc_1, \ldots, uc_U\}$, 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):

$$\begin{aligned} UC &= \{uc_1, \ldots, uc_U\} \\ &= Cluster(PreBZ^{Concept}, Rand^{Concept}) \end{aligned} \tag{4.11}$$

$$\begin{aligned} uc_i &= (center_i, label_i), \\ label_i &\epsilon \{"nonBZ", "preBZ"\} \end{aligned} \tag{4.12}$$

Figure 4.5 illustrates an abstracted example drawn in two dimensions. There are four UCs, 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 of members. Based on this abstraction image, we describe the Bug-Zone prediction functionalities.

4.3.3 Log Mining Tasks

4.3.3.1 Online ML-based Bug-Zone Prediction

Online Bug-Zone prediction gives an advance warning to system administrators about imminent anomalies and probable system failure. As we discussed in chapter 3 to have an online predictor, we can simply train a classifier, here 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.

In addition, in chapter 3 we proposed a second approach to use the created UCs centers as indicators to determine if a sequence of events may cause Bug-Zone. Creating UCs and calculating cosine similarity allows us to determine whether a new sequence of input test events can lead to *Bug-Zone*. For instance, we imagine that the smallest cosine distance is between the new sequence

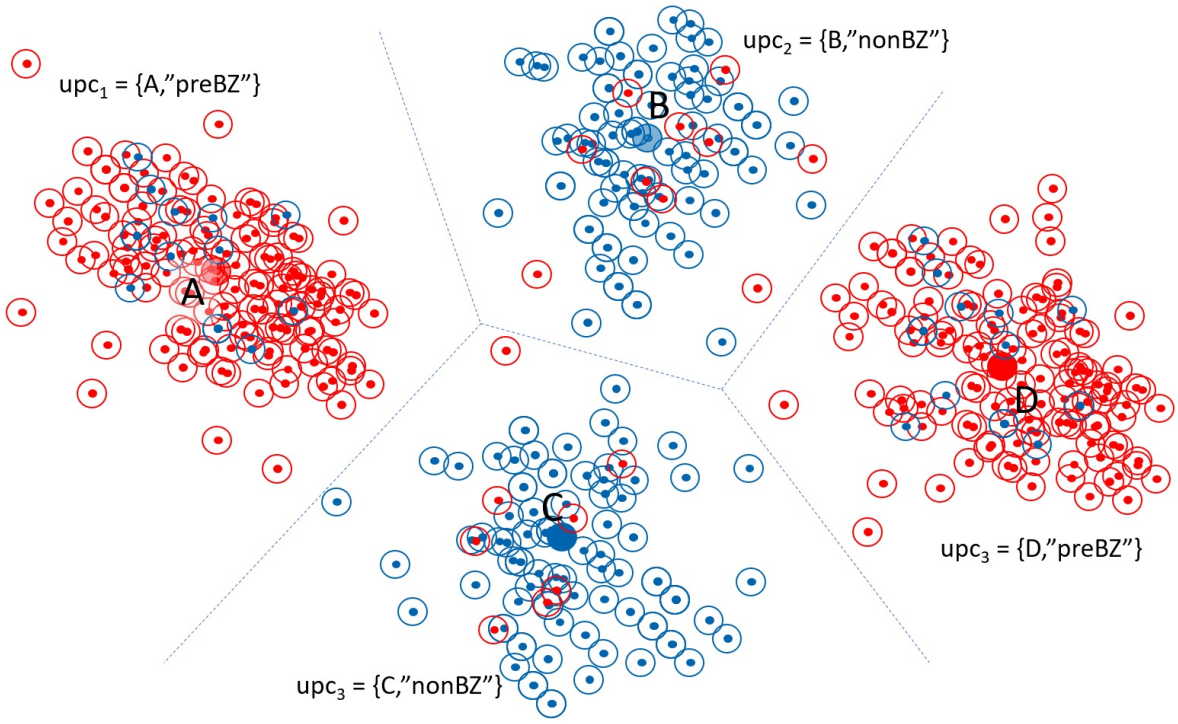


Figure 4.5: Universal Clusters (UC) - Each dot is a sentence processed in the learning phase (Red: Pre-Bug-Zone, Blue: Random-Zone)

of input test and a Pre-Bug-Zone UC. Therefore, we predict a Bug-Zone to happen soon. Then, by a new input arrival, the cosine distance must be calculated again to find the closest UC.

4.3.3.2 Root-Cause Detection

Root-cause detection aims to find the events that are associated with bug occurrences and to this aim, the UCs do convey important information to be extracted. The theme of this analysis is to learn which *concept* differentiates a bug UC from a normal UC. We can take two approaches to do so: 1) UCs subtraction 2) Logistic regression.

UCs subtraction: On Figure 4.5, two UCs A and B are Pre-Bug-Zone and Random-Zone, respectively. By calculating the A-B subtraction, one obtains a vector like $[diff_1, \ldots, diff_C]$. The index of the largest value (e.g: f_{imax}= max $[diff_1, \ldots, diff_C]$.) indicates the concept with the highest contribution in the *Bug-Zones*. Therefore, the members of $Concept_{imax}$ from the equation are the potential root cause of the anomalies, since their presence differentiates members of A from B. It should be recalled that the members of $Concept_{imax}$ are words that are equivalent to input types that directly contribute to creating anomalies.

Logistic regression: Logistic regression is a common Machine Learning method that belongs to the Supervised Learning technique. It is used to forecast the categorical dependent variable from a group of independent variables. We can exploit a model created by logistic regression to extract *feature importance*. It determines which features are truly important in predicting your target variable. Using the model's coefficients is the simplest technique to calculate feature importance in binary logistic regression. The coefficients represent the log odds change for a one-unit change in the predictor variable. Greater absolute values suggest a more significant association between the predictor and the target variable. Therefore, the feature with the highest coefficients represents the root cause concept and since the concepts are a group of event types, the bug root cause must be among the event in the concept with the highest coefficient.

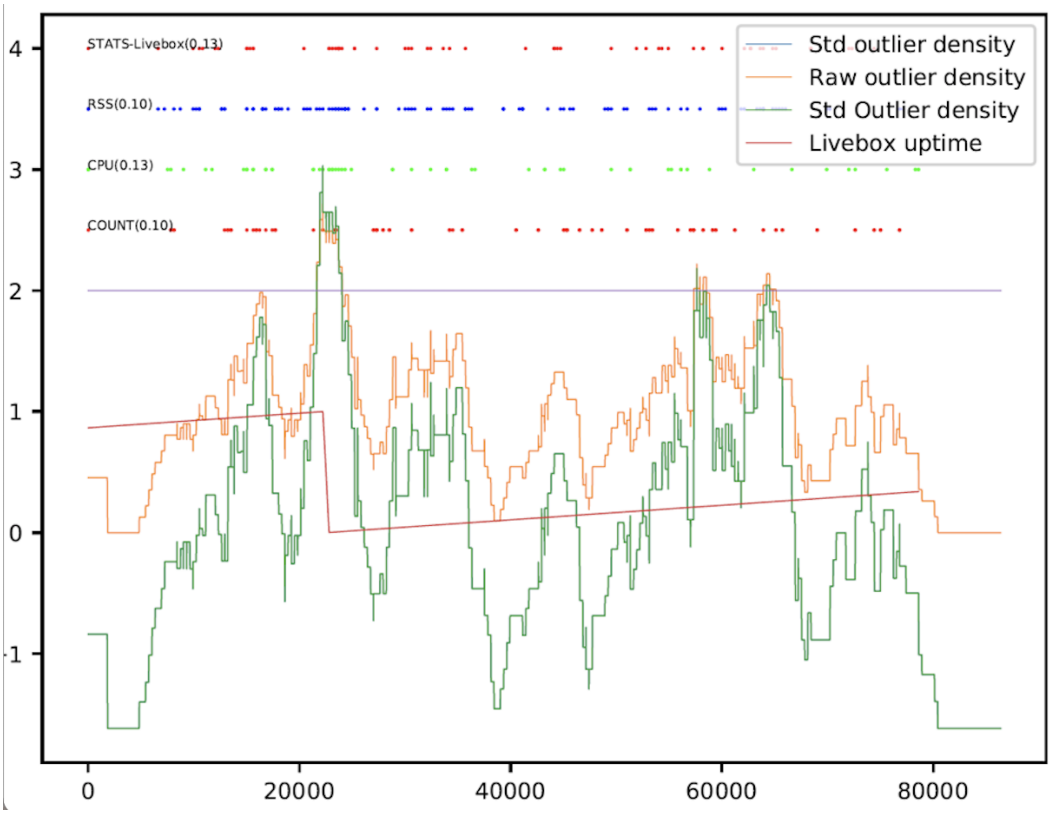


Figure 4.6: Outlier density curve and detected *Bug-Zones* in the Telecom case study

4.4 Implementation and Evaluation Results on Prediction and Root-cause Detection

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

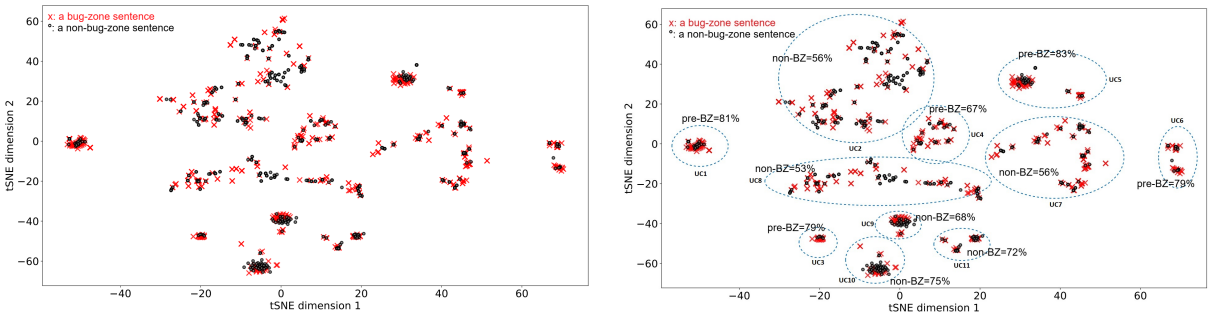
*Q*1: Can our model distinguish Pre-Bug-Zone from Random-Zone sequences accurately enough?
*Q*2: How effective is the proposed approach in predicting *Bug-Zones*?
*Q*3: What is the complexity of the proposed approach?
*Q*4: What are the accuracy and efficiency of our proposed method in root-cause detection?

4.4.1 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 with two outlier detection methods, Local Outlier Factor and Isolation Forest [145, 169]. 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 [170].

4.4.1.1 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,



(A) (B)

Figure 4.7: (A) A projection of Pre-Bug-Zones and Random-Zones vectors in 2D from the telecom case study,(B) UCs created from telecom case study

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 4.6 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 proportion is in line with our observations). The high correlation observed, which lies between 70% and 100% (although the absence of further data prevented us from computing a more accurate value) indicates that the Bug-Zone finder is effective in finding anomalous behavior 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 behavior that 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 input 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 [171], 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 4.7(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 the Elbow method [171] to find the number of clusters. The plotted image of the UCs in the telecom case study is depicted in Figure 4.7(B). In this figure, we can distinguish some UC clusters in which either red crosses or dark dots are in the majority. The more imbalanced colors in each UC are, the more meaningful the cluster is. For instance, UC1 has a very high number of Pre-Bug-Zone (red) crosses 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 UCs, UC2 and UC8 in which both Random-Zone and Pre-Bug-Zone sentences appear almost equally in these clusters. For the sentences that fall inside these UCs, 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.

4.4.1.2 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 4.8 by red vertical bars.

The outlier density curve is depicted in Figure 4.9. 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 4.9, 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 4.10-A shows a projection of the averaged sequence vectors in 2D space, while Figure 4.10-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 vector-

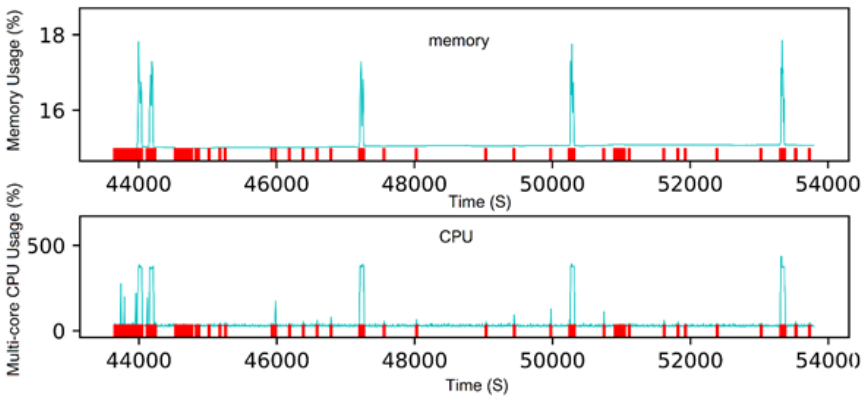


Figure 4.8: CPU and memory usage during the test period and five bug events in the Train Ticket case study. Detected anomalies are depicted by red bars.

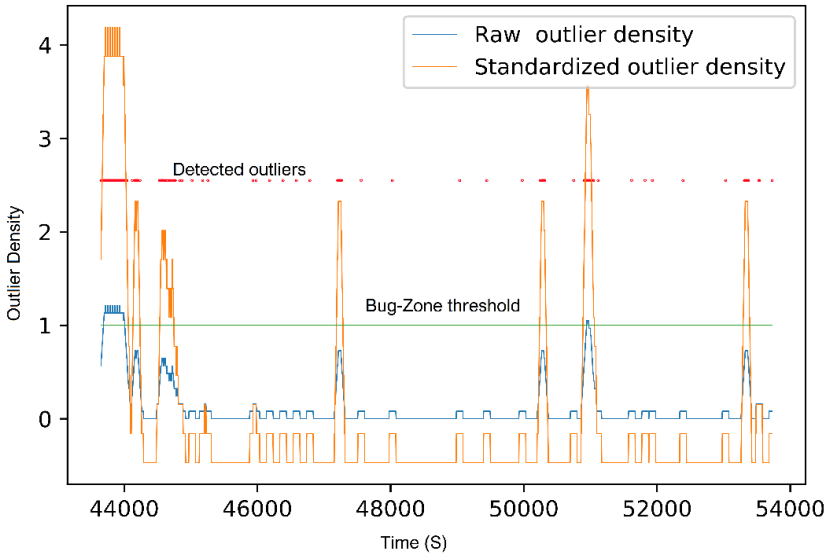


Figure 4.9: Outlier density curve for the Train Ticket benchmark. The detected outlier is shown in red dots and the threshold line is in green.

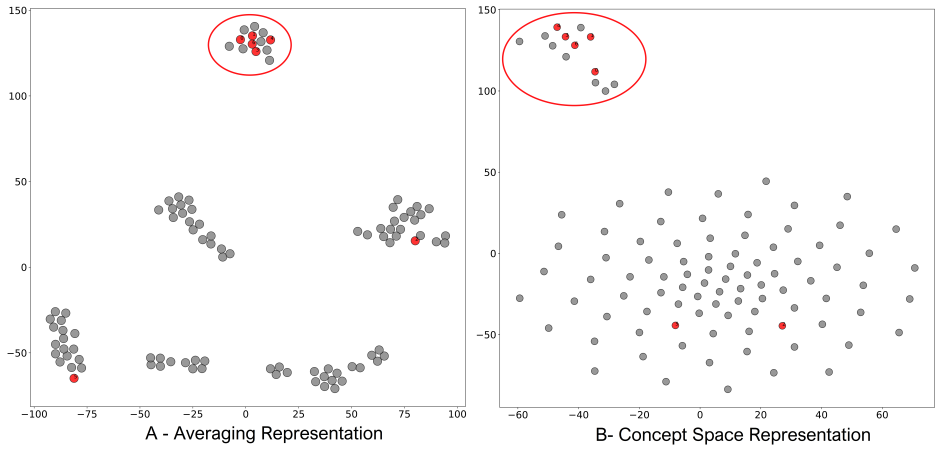


Figure 4.10: Averaging method vs concept space method to sentence embedding in the Train Ticket case study

ization 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 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 be based on an estimation of the false positive rate of Bug-Zone prediction. For instance, in Figure 4.10-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 4.10-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%

Table 4.1: Classification methods applied on Pre-Bug-Zone and Random-Zone sequences on the Telecom Dataset

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

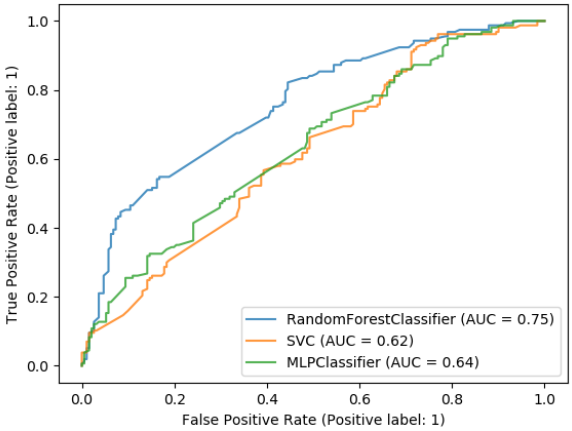


Figure 4.11: The Roc curve for Random Forest, SVM, and MLP classifiers for Telecom case study

accuracy in predicting *Bug-Zones*. A higher precision was expected in this case study due to the lower noise and complexity of the system.

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

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 in classifying the Pre-Bug-Zone and Random-Zone sequences is presented in Table 4.1. Random Forest, with 75% accuracy, has the highest rank.

AUC metric or Area Under Curve value [172] is one of the most important metrics for evaluating any classification model's 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 in Figure 4.11 show that the Random Forest classifier outperforms the other ones, since the AUC value for Random Forest is 0.71, while the AUC value for SVM and MLP classifier is 0.62 and 0.64, respectively.

Table 4.2: AUC values in different dataset

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

4.4.3 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*.

4.4.3.1 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 result. 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 the telecom case study and 90.7% accuracy in predicting *Bug-Zones* in the Train Ticket dataset. These results imply that it can be used to predict *Bug-Zones* based on real-time incoming test data. A higher value was expected in the Train Ticket benchmark due to the lower noise and complexity of the system.
Table 4.2 shows these results. By using UC prediction method, we achieved 65% and 95% accuracy in Telecom and Train Ticket case studies respectively.

4.4.3.2 Precision, recall and F1-score

We computed common classification metrics, namely, *precision*, *recall*, and *F1-score* which are routinely used in similar work [63] [167] [173] [71] 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 [173], 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 tester's point of view, *Recall* metric might be more important, since a lower False-Negative rate (or higher True-

Table 4.3: 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	UC Prediction	0.63	0.62	0.61
Train Ticket	RF classifier	0.93	0.93	0.93
Train Ticket	UC Prediction	0.97	0.75	0.81

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.

4.4.3.3 Comparative Analysis: What to choose: Random Forest or UC Prediction?

Table 4.3 shows the precision, recall, and F1-score on the telecom and Train Ticket case studies by using the Random Forest classifier as a baseline and UC prediction as a proposed method in this thesis. It shows that UC prediction can achieve higher precision values (e.g. 97% for Ticket Train). On the other hand, it has lower recall values and, therefore, a higher false negative ratio.

To answer the "what to choose" question, we must state that, in general, Random Forest shows better efficiency (F1-score) specifically in noisy log files (e.g., Telecom case study). Then, if efficiency is the only parameter, RF is the way to go. However, the UC cluster has two main advantages that may tip the balance in its favor in many case studies:
- *Sustaining unsupervised approach:* Every stage in the suggested process upholds the commitment to offer an unsupervised approach. RF is a supervised classifier that requires a training phase. However, it won't have the exact difficulties of a supervised classifier in our case. the learning phase does not need a manual intervention to label the learning dataset, because the learning set is already labeled (Pre-Bug-Zone and Random-Zone) during the model creation phase.

- *Complexity on model creation and prediction*: In general, UC prediction has considerably lower computational complexity. RF has several parameters that contribute to its construction complexity. A full unpruned decision tree takes O(v * n log(n)) time to construct, where v is the number of variables/attributes and n is the number of records. One must specify the number of trees to be built (assuming it to be, ntree) and the number of variables they wish to sample at each node (assume it to be, nvar) before beginning to build random forests. The complexity to construct a single tree would be O(nvar * n log(n)) since we would only employ nvar variables at each node. Now, the complexity for creating a random forest with ntree trees would be O(ntree * nvar * nlog(n)), assuming the tree's depth will be O(log n), and this is the worst-case scenario. However, a tree's build typically ends considerably earlier, making an estimation difficult. Nevertheless, we could also limit how deep the trees could grow. The complexity calculations can be simplified to: If we limit the maximum depth of our tree to be "d": O(ntree. nvar. d. n). In contrast, UC prediction is only a cosine distance calculation at each prediction. Therefore, for large datasets and real-time applications, UC prediction has the advantage of faster execution time and lower memory footprint.

4.4.4 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 input 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 UCs, 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 UCs. Therefore, using UCs is preferable due to lower complexity in real-time systems.

4.4.5 Q4: What are the accuracy and efficiency of our proposed method in root-cause detection?

As described in subsection 4.3.3.2, the distance between a Bug-Zone universal cluster (UC) and a random (normal) UC can be exploited to learn which concept (a set of related tests) contributed more to creating that Bug-Zone UC. As was discussed, one has two ways to find bug root causes. 1) It is possible to calculate the distance between a Pre-Bug-Zone UC and a normal UC by subtraction and find the most important concepts as the maximum absolute value after the subtraction. or 2) use logistic regression and extract feature importance coefficients. We chose the second method since it was easy to deploy using sklearn Python library.

We trained a supervised logistic regression model with the Pre-Bug-Zone and random-Zone vectors. After model creation, we extracted the model coefficients. Among the 20 concepts created in subsection 4.4.1.1, five had a coefficient higher than 0.5. In particular, $Concept_4$, $Concept_1$, $Concept_5$, $Concept_{18}$ and $Concept_{17}$ have coefficients of 0.70, 0.69, 0.63, 0.55, and 0.51, respectively. $Concept_4$ is comprised of 7 input events (4% of the total of 175 input events), and all five concepts, together, are comprised of 37 test input events which form 21% of the total test events. Therefore, now the telecom's developers have a smaller number of input events to study and realize which subset of them triggers the Bug-Zones.

To verify the validity of the detected root-cause input test events from the Telecom case study, we must run the suspected test events again on the Telecom device and observe its health status to see if the probability of triggering anomalous behavior increases. We may succeed in driving the device into failure if the root-cause events are valid. However, we didn't have access to the test bench to re-examine the proposed root-causes. We submitted a report of the suspected fault causes to the Telecom company and asked them to do so. We are still waiting for the response from their test team.

4.5 Threats To Validity

The main threat to the validity of this study is that the approach has only been validated in two case studies, with different contexts and scales. Obviously, it should be assessed in 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 by a previous study [81] 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 failure. We would need to inject other types of failures in the future.

Another threat to the validity of our study, 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.

The Last one, there is 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.

4.6 Conclusion and Future Work

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 chapter 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 input 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 UC 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 UC predictor.

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

Chapter 5

Log Minimization: Scanner case study

Contents

This chapter has been published as a paper in the 1st Workshop on Natural Language Processing Advancements for Software Engineering (2020, Singapore) [10]. It covers the scanner (scanette in French) case study, which employs the proposed method for a different purpose than the previous case studies. The goal of log mining in this case study is to minimize huge (simulated) usage logs into tiny ones while keeping the fault-triggering capabilities of the whole logs. Obviously, executing the whole test suite and selecting the most efficient tests is the worst-case scenario and requires a very long and tedious effort. Therefore, any effort to select a small number of tests without executing the whole test suite saves time, reduces costs, and is appreciated by software testers.

5.1 Introduction

Software testing is one of the most time-consuming and highly priced steps in software development, specifically for large systems. It accounts for more than 52 percent of the total software development budget, and its fault detection coverage is directly connected with the quality assurance of the product [174]. Therefore, any effort in optimizing the test process can save time and budget and increase product quality. These, in turn, will ease software testing of large systems, shorten the time-to-market gap and increase the profit.

During the software testing process, the product may go through regression tests, which include several recurring test-and-debug steps to ensure that the software still performs after modification with the same functionality as originally. Regression testing is crucial to ensuring the quality and reliability of a software product. Regression testing requires considerable time and development resources. Due to the addition of new features, test engineers must generate a new set of test cases in order to validate the modified portion of a software system. As a consequence, some test cases become obsolete and redundant, which requires additional testing time and resources. A large number of test traces can be gathered either from automated testing or from user traces. Test Suite Minimization or Test Suite Reduction (TSR) helps to reduce and purify the test cases by removing redundant and ineffective tests [175]. Therefore, test suite reduction is a method for accelerating regression testing. The objective is to determine which tests can be eliminated from a test suite without significantly diminishing its fault-detection capability. Numerous algorithms have been proposed by researchers to identify redundant tests [175, 176, 177]. An automated TSR helps to reduce human interaction in error case debugging since the process leaves fewer test cases to be investigated by humans. During the last decade, several automated TSR methods have been proposed and recently Machine Learning (ML) has extended its realm to software testing and test suite reduction.

In the literature, there are mostly three main approaches employed for TSR; Greedy, Clustering, and Search approaches [177, 178]. These approaches mainly differ from each other in terms of the type of algorithms used. Greedy-based approaches mostly use greedy algorithms, clustering-based approaches use a variety of clustering and sampling algorithms, and search-based approaches employ a variety of search algorithms. The existing clustering-based TSR approaches employ supervised clustering algorithms to group test cases. Applying clustering-based approaches for TSR has received a deal of attention [178, 179, 180]. Some similarity-based approaches have been proposed for clustering, which generally tries to find similar test cases and remove redundancy [181]. Reichstaller *et al.* [182] used two clustering techniques, *Affinity Propagation* and *Dissimilarity-based Sparse Subset Selection* to reduce the test suite in a mutation-based scenario. Felbinger *et al.* [183] employed decision trees to build a model from the test elements and remove those that do not change the model. In addition, classic machine learning approaches (*e.g:* Random Forest and SVM) are employed for this purpose [184].

Furthermore, clustering can be utilized to extract knowledge from user traces. By clustering

user traces, the most important behaviors of the users can be extracted. The authors in [185] presented a user session-based testing technique that clusters user sessions based on the service profile and selects a set of representative user sessions from each cluster. Following that, each selected user session is tailored by increasing it with additional requests to account for inter-webpage dependencies.
The mentioned methods generally focus on the combination of the test elements and do not consider the order and vicinity of the elements in a test trace. Since our proposed method applies NLP to create an ML model, it effectively takes into account the combination and order of the events in test traces. For this purpose, the proposed method processes test traces as sentences in a natural language and builds a model based on the sequence of the words in each sentence.

In contrast to many TSR methods that need to run the test traces to decide on keeping or removing them, the proposed method regards the traces like natural language sentences, by separating and removing similar and redundant events. Therefore, it does not need to run the traces, which eliminates the need to access the software-under-test and makes the entire test reduction process faster. In addition, our methods employ unsupervised clustering algorithms for grouping sessions.

In this chapter, we adopt the proposed method in chapter 3 for TSR purposes with an approach to identify semantic similarities between individual test sentences and choose one test sentence as a representative of a subset of similar sentences. More specifically, in section 5.3 we will explain the steps for applying the proposed method to test suite minimization. We will introduce the scanner case study in section 5.4. Then, we will discuss the results of the proposed method in section 5.5 as well as the conclusion and future works in section 5.6.

5.2 Assessment of approach: Mutation testing

Mutation testing is a technique that has been widely used in academic research. It often used as a "gold standard" to compare testing approaches [186]. This technique is centered on the concept of changing the System Under Test (SUT) in such a way that the modifications simulate faults that a component programmer would produce. The generated mutations of SUT are referred to as *mutants*. After that, testers would design test cases for discovering the seeded faults. When a test case enables a mutant to act differently than the original SUT, it is said that the test case *"kills"* the mutant. The hypothesis underlying mutation testing is that a well-designed test suit can identify all mutants. Thus, testers must improve the test suit until it can kill mutants that are not identical to the original SUT. A defective test suite is one that does not detect certain mutants. The goal when applying mutation testing is to obtain a mutation score of 100%. It indicates that the test suit is able to detect all the faults represented by the mutants [186].

Mutation testing is a highly effective way to evaluate the effectiveness of a test suite, but it comes with a large cost. Basically, manually carrying out mutation testing requires a lot of human effort. Since it is costly and time-consuming, researchers have been trying to use ML algorithms to accelerate some steps of this process. There are two main approaches to reduction, which can be categorized into two groups: the first consists of methods aimed at decreasing the number of mutations, and the second, second, those aimed at reducing the execution costs. For instance, in [187] the similarity of mutants, is used to reduce the number of mutants to be executed. Graph representations of each mutation are used to determine their similarity. The use of graphs to represent objects has been the focus of much research. [187] proposes a method for classifying mutants as a way to reduce the number of mutants to be executed and evaluate the quality of test suits without exposing them to execution against all available mutants.

We evaluate our proposed method on a Supermarket Scanner (Scanette) case study. In order

to assess the fault coverage of a reduced test suite, we use mutation-based testing, which has become mature and is being applied more and more both in research and industry [188]. Based on this approach, the source code of the Scanette software is artificially mutated. For that software, we had a source of handmade mutations that were known to provide a good assessment of fault coverage. Each mutation injects a bug into the Scanette software. For that case study, we also had three reference test suits of varying lengths, one of them from a (handwritten) functional test suite, and the others collected from random testing. We apply our method to assess its results in that case study.

5.3 Applying the Proposed Method

Here, we explain the steps for applying the proposed method to log minimization. The input of the proposed method can be a test suite or usage logs comprised of several clients' sessions, each of which contains a trace of the client's actions. The goal is to remove redundant and ineffective sessions to have a considerably lower number of sessions that have the same effect as the original test suite or usage log. Since the input file can be either a test suit or a usage log, we consider test or event selection as a log minimization task.
we adopted the general proposed method from chapter 3 and modified it to achieve the TSR task. We pursue the steps depicted in Figure 5.1, which will be discussed in turn: log pre-processing, model creation by the NLP in which we have vectorized session representation, and Universal cluster creation, then in order to reduce the number of test cases for log mining task, we select representative sessions, and finally, there is execution and verification step. The

5.3.1 Pre-Processing

The pre-processing step has a few steps to create a dataset for ML model creation. These steps are log parsing, log partitioning, and event abstraction. In an initial log parsing and partitioning step, we cut a trace that records interleaved sessions of different independent customers into a set of sessions, each one for a unique customer. Then, as an event abstraction step, each event of a trace is converted into a triplet that captures the relevant features of the event.

A session refers to a sequence of actions performed by a single user while interacting with the system. As the session is a sequence of events, after conversion it becomes a sequence of triplets. In the event abstraction step, we describe the association of triplets to events. We decided to differentiate among similar events which have different inputs and outputs because every combination of action-input-output may show different behavior of the system. For example, a 'scan' action with an error output code should be treated differently from the same action with a success output code. For this purpose, each action-input-output was coded as a triplet vector like $[a, p, o]$, in which 'a' is the index of the action from set A which itself includes n possible actions denoted by A_1 to A_n. Likewise, 'p' is the index of the input parameter from set P, containing all possible input parameters from P_1 to P_m and finally 'o' is the index of the output parameter in set O including a list of all possible outputs, from O_1 to O_k.

$$A = \{A_1, A_2, A_3, \ldots, A_n\},$$
$$A_1 : unlock, A_2 : scan, A_3 : add, \ldots$$
$$P = \{P_1, P_2, \ldots, P_m\},$$
$$P_1 : barcode, P_2 : null[], P_3 : floatnumber, \ldots$$
$$O = \{O_1, O_2, \ldots, O_k\}$$
$$O_1 : 0, O_2 : -2, \ldots$$

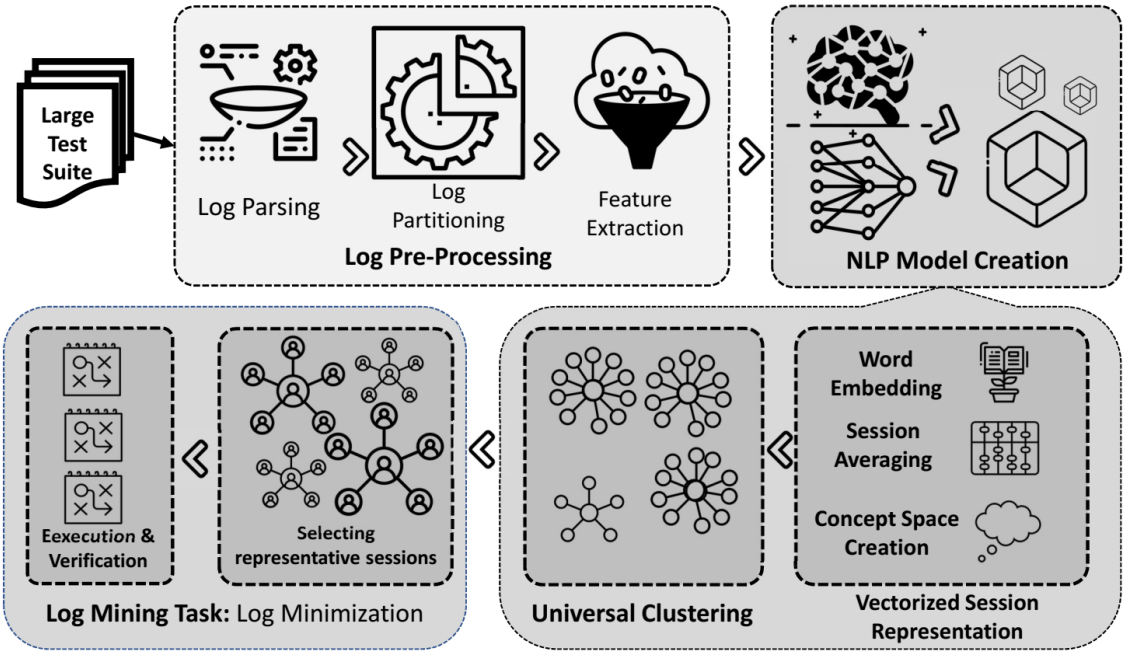


Figure 5.1: Flow chart of the proposed session reduction approach on the scanner case study

Therefore, we can encode all actions/inputs/outputs in triplets. Note that we still abstract from the timestamp and from the parameters of the action (typically, the actual barcode of a shop item): the actual delay between actions is not relevant, only the ordering of events should be kept. Similarly, the exact choices of items picked by a customer are irrelevant to the logic of the application. We can now display each session as a sequence of triplet vectors. You can see three different clients' sessions below. Each session has been printed in the form of triplets: **client 60**: [['debloquer', 'Nothing', '0'], ['scanner', 'Barcode', '0'], ['scanner', 'Barcode', '-2'], ['scanner', 'Barcode', '0'], ['transmission', 'CaisseNumber', '0'], ['abandon', 'Nothing', 'Error'], ['ouvrirSession', 'Nothing', '0'], ['ajouter', 'Barcode', '0'], ['fermerSession', 'Nothing', '0'], ['payer', 'Price-float', '0']]

client 40: [['debloquer', 'Nothing', '0'], ['scanner', 'Barcode', '0'], ['scanner', 'Barcode', '0'], ['scanner', 'Barcode', '0'], ['scanner', 'Barcode', '0'], ['scanner', 'Barcode', '0'], ['scanner', 'Barcode', '0'], ['scanner', 'Barcode', '0'], ['transmission', 'CaisseNumber', '0'], ['abandon', 'Nothing', 'Error'], ['payer', 'Price-float', '0']]

client 17: [['debloquer', 'Nothing', '0'], ['scanner', 'Barcode', '0'], ['scanner', 'Barcode', '0'], ['scanner', 'Barcode', '0'], ['scanner', 'Barcode', '0'], ['scanner', 'Barcode', '0'], ['scanner', 'Barcode', '0'], ['scanner', 'Barcode', '0'], ['scanner', 'Barcode', '-2'], ['scanner', 'Barcode', '0'], ['transmission', 'CaisseNumber', '0'], ['abandon', 'Nothing', 'Error'], ['ouvrirSession', 'Nothing', '0'], ['ajouter', 'Barcode', '0'], ['fermerSession', 'Nothing', '0'], ['payer', 'Price-integer', 'Float Number']]

5.3.2 Model Construction

In this section, we describe model construction that consists of: word representation, vectorized session representation, universal cluster creation, that will be covered in order. First, we associate a vector to each triplet using the Word2Vec vectorization process, as described in section 5.3.2. The dimension of those vectors is the number n of different triplets that appear in the trace. Each session of length L is therefore associated to a sequence of L vectors of size n. Finally, we associate a single vector of size n to a session by using session averaging, as described in section 5.3.2.2. But as n would be too high a dimension in most cases, we first reduce the dimensionality using the t-SNE method before proceeding to the clustering. We go into the details of model creation in the

following subsections:

5.3.2.1 Word Representation

In the continuation of the proposed method from chapter 3, an NLP method must create a vector representation of the test sequences. The output of this approach is a vector representation of each word. In different studies, it has been shown that the distance between each pair of words translates their semantic relation. From word representation, a representation of each sentence in a given document of the collection can be induced by for example averaging the vector representations of all words that are present in the sentence. In our work, since we need to cluster/merge similar sessions into a fewer number of sessions that can trigger the same errors (or "kill the same number of mutants"), we chose Word2Vec to find semantically similar sessions by treating test traces as sentences. The Word2Vec clustering method may tell us that some sessions are equivalent or very close to each other, although their actions and inputs/outputs are different. For this purpose, first, we calculate a measure for each triplet and use the Word2Vec method to learn its vector representation. It should be noted that we store each triplet as a string (e.g: '['scanner', 'Barcode', '0']') and we treat them like words in natural language processing.
These are two 15-element vectors created by Word2Vec method and represent ['scanner', 'Barcode', '0'] and ['delete', 'Barcode', '1'], triplets.

```
[['scanner', 'Barcode', '0'] :
[ 1.2445878, 1.613417, -0.1642392, 3.0873055, -0.355896 , 1.0599929, -0.49392796,
1.0838877, -1.1861929, -0.2639794, -0.09810112, -0.9824149, 0.881457, -3.6238787,
-1.1903458 ]

['delete', 'Barcode', '1'] :
[ 0.17494278, 0.15232983, -0.14811908, -1.5120562, -0.0818198, -0.35962805,
-0.65130717, -0.18931173, 0.85284257, -0.23423576, 0.8646087, 0.41952062, 0.5157884,
1.593384, 0.50375664]
```

5.3.2.2 Vectorized Session Representation

Similar to the Telecom case study, we create a single vector to represent a sequence of input vectors. There are different methods to get the session vectors [138, 141, 155, 156]. In fact, a common method to achieve sentence representations is to average word representations. Vector averaging has been effective in some applications [189]. Averaging over word vectors in a sentence was shown to be an effective method for sentence representation. The authors in [190] studied three supervised NLP tasks and observed that, in two cases including sentence similarity, averaging could achieve better performance in comparison to LSTM. To this end, we compute an average of the Word2vec vectors in each session. It could be simply an element-wise average of n-element vectors that finally leaves us an n-element average of the words in a sentence. In the end, we have a single vector measure for each session. In our experiments, we used the element-wise arithmetic mean to compute the averaged measure for a session. Hence, each session is associated to a single vector of size n.

Concept space creation, as covered in previous chapters, is another method to achieve this goal. Since we achieved satisfactory results by using the averaging method in this case study, we did not feel motivated to apply the concept space approach. We left the concept space creation as a part of the future work.

5.3.2.3 Universal Clusters Creation

As described in the chapter 3, we create universal clusters from the sentence representations. This is an essential step before log mining tasks. In this case study, universal clusters have no label since there is no binary distinction among the clients, unlike the case studies in subsubsection 4.4.1.1 with "Pre-Bug-Zone" and "Random-Zone" labels.

Traditional approaches to data analysis and visualization often fail in the high dimensional setting, and it is common to perform dimensionality reduction in order to make data analysis tractable and meaningful [191]. To this end, we applied the t-SNE algorithm [153] in order to reduce the initial dimension of the vector space to 2, as it has been proved that this method successfully maps well-separated disjoint clusters from high dimensions to the real line to approximately preserve the clustering. Also, dimension reduction is shown to be effective in some clustering case studies [153]. We will show the effect of the dimension reduction on the results later in the next Section.

After dimension reduction, we apply the K-means algorithm for clustering the sessions in the reduced space of dimension 2 and employ the Elbow technique [171] to estimate the optimal number of the universal clusters. Since t-SNE and K-means choose random initial points, we repeated each experiment two times and chose the best result.

5.3.3 Log Mining Tasks: Log Minimization (Reduction)

The number of user sessions for an application that has been in production for a long period of time can be very high. Using all the obtained user session data requires significant effort to determine which portion of the data most accurately describes the system's behavior. Nonetheless, a significant amount of user sessions does not naturally guarantee sufficient coverage of the system's expected behavior. Test selection from usage logs is the main goal of this case study. The approach is to select a representative session from a cluster of sessions in which all the sessions have similar semantics and similar bug-triggering capabilities. Some steps are required to be taken that will be covered here:

5.3.3.1 Representative Selection

For test selection, we decided to keep one representative from each universal cluster and see how many mutants they can kill. For this purpose, we select from each UC, the client with the highest number of events (the longest session). Experimentally, this client kills more mutants than shorter ones, as can be expected.

Alternately, representative clients (sessions) can be selected based on various criteria (besides the longest session). For instance, we can select the closest client to the center, the shortest session, or a session with the most diverse events.

5.3.3.2 Execution and Verification

This is the final step to know if the selected representation has the same bug-triggering capability. We can evaluate our proposed method for test suit reduction on killed mutants. The reduced test suite has to kill the same mutants killed by the original test suite. Here we are allowed to execute them since their number is quite small in comparison to the complete test suite. By observing the outputs during their execution we can understand how many of the bugs are covered by this small subset of test sessions.

5.4 The Software Under Test

A: Definition

We define a user trace as a sequence of API calls by which the user has triggered the system. We may refer to it as an Event. A user trace includes time stamps, several parameters, user identifiers, API methods or actions that the user does, and the output of the system. The experiments presented in this chapter were performed on Scanner case study.

B: Scanette case study (scanner)

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

The Scanette system has a Java implementation for development and testing and a Web-based graphical simulator for illustration purposes. The web-based version emulates customers' shopping and self-service check-out in a supermarket by a randomized trace generator derived from a Finite-State Machine. The trace logs of the Scanette system contain interleaved actions from different customers who are shopping concurrently. Each customer has a unique session ID which distinguishes his/her traces from another customer. Figure 5.2 shows a snippet of a trace log with different actions from different sessions. Each event in the test (trace) has an index and time stamp which are stored chronologically. The Session ID determines to which user (or session) the event belongs. In this figure, we can find activities of *Client6*, interleaved with other clients' actions, which start with an 'unlock' and end by a 'pay' action. The triplet of action-input-output obviously shows which action is called, what is given as its input, and what is obtained as the output of the action. Some actions may also include a parameter, such as the actual value of the barcode of an item that is scanned.

To artificially inject faults, the source code of the Scanette software is mutated with 49 *mutants*, all made by a modification on the source code by hand.

We needed a few logs to be used as the test bench for the proposed method. The test bench of 1026, 100043, and 200035 events was provided by Philae. We will call them as 1026-event, 100043-event, and 200035-event names, respectively. They include shopping steps for different numbers of clients (sessions). They were created as random usage logs by a generator of user traces that simulate the behavior of customers and cashiers. The goal of the proposed TSR methods is to reduce the number of traces needed to kill the same mutants as the original test suite can kill. In the rest of this chapter, *session* and *client* are equivalent.

Index	Time	Session ID	Object	Action	Input	Output
51,	1585070116817,	client6,	scan12,	unlock,	[],	0
52,	1585070116819,	client0,	scan0,	scan,	[3270190022534],	0
53,	1585070116820,	client1,	cashier1,	CloseSession,	[],	0
54,	1585070116820,	client2,	cashier2,	add,	[3570590109324],	0
55,	1585070116824,	client5,	scan5,	scan,	[8718309259938],	0
56,	1585070116825,	client6,	scan12,	scan,	[3560070139675],	0
57,	1585070116837,	client0,	scan0,	scan,	[3560070048786],	0
58,	1585070117030,	client6,	scan12,	scan,	[7640164630021],	-2
59,	1585070117073,	client6,	scan12,	delete,	[7640164630021],	-2
60,	1585070116838,	client1,	cashier1,	pay,	[353.06],	0
61,	1585070116839,	client2,	cashier2,	CloseSession,	[],	0
62,	1585070116840,	client3,	cashier3,	add,	[3570590109324],	0
64,	1585070117687,	client6,	scan12,	transmission,	[caisse6],	0
65,	1585070117687,	client6,	scan12,	abandon,	[],	?
66,	1585070117701,	client6,	cashier4,	OpenSession,	[],	0
67,	1585070116855,	client0,	scan0,	transmission,	[cashier0],	0
68,	1585070116855,	client0,	scan0,	abandon,	[],	?
69,	1585070117716,	client6,	cashier4,	add,	[7640164630021],	0
70,	1585070117731,	client6,	cashier4,	CloseSession,	[],	0
71,	1585070117747,	client6,	cashier4,	Pay,	[260],	9.11

Figure 5.2: Test traces of the Scanner case study

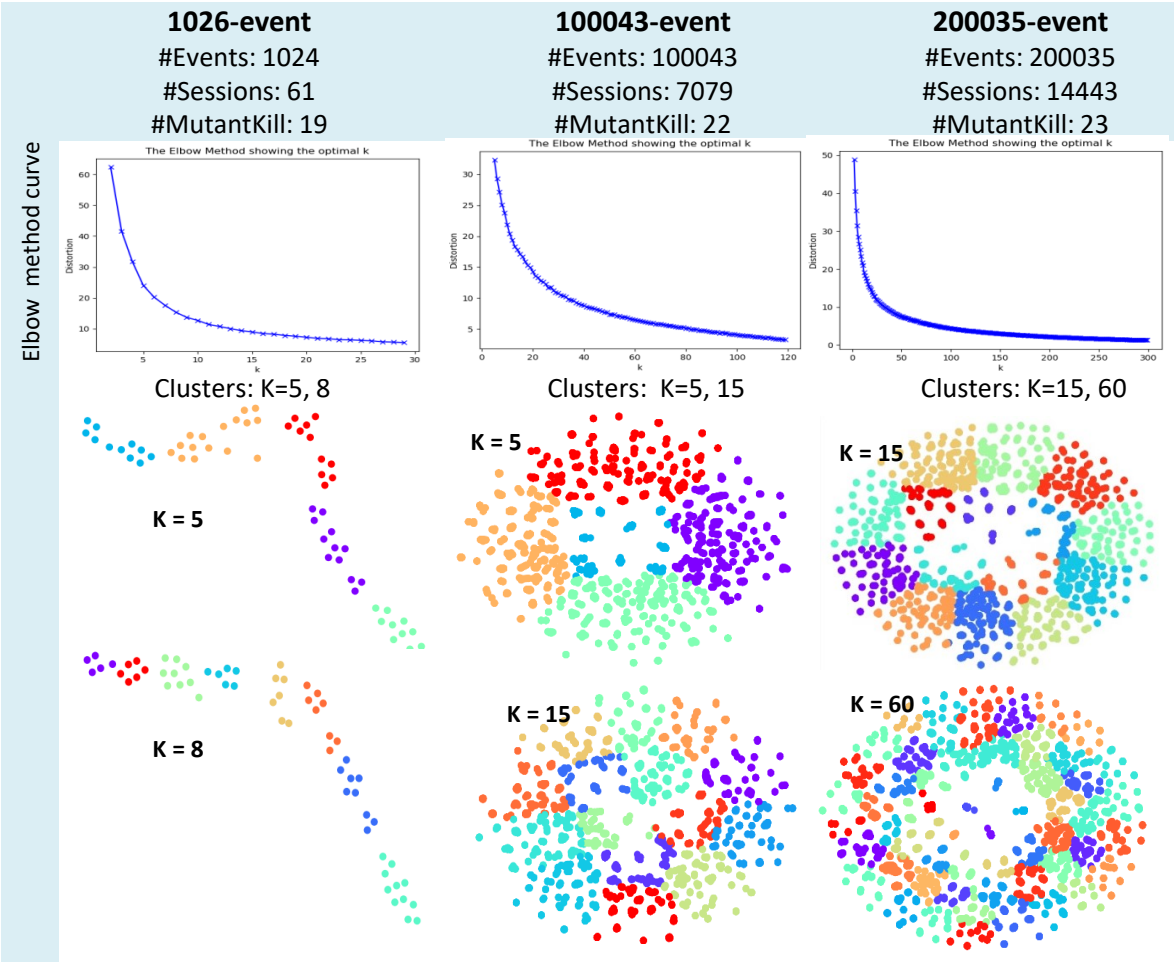


Figure 5.3: Elbow method curve and clustering visualization in finding optimal number of universal clusters

Table 5.1: Optimal number of clusters for the 1026-event Scanette case study

K	#Events	Sessions Numbers	Killed mutants	Killed Mutant IDs
2	36	[30, 24]	9	[7, 12, 17, 19, 20, 21, 25, 26, 42]
3	67	[30, 23, 35]	17	[0, 7, 9, 10, 12, 17, 19, 20, 21, 25, 26, 34, 35, 37, 41, 42, 44]
4	81	[30, 6, 10, 24]	10	[7, 12, 17, 19, 20, 21, 25, 26, 42, 48]
5	104	[30, 6, 10, 24, 23]	18	[0, 7, 9, 10, 12, 17, 19, 20, 21, 25, 26, 34, 35, 37, 41, 42, 44, 48]
6	117	[30, 28, 23, 6, 24, 10]	18	[0, 7, 9, 10, 12, 17, 19, 20, 21, 25, 26, 34, 35, 37, 41, 42, 44, 48]
7	128	[6, 27, 23, 10, 24, 30, 28]	18	[0, 7, 9, 10, 12, 17, 19, 20, 21, 25, 26, 34, 35, 37, 41, 42, 44, 48]
8	147	[27, 52, 23, 10, 24, 6, 30, 28]	18	[0, 7, 9, 10, 12, 17, 19, 20, 21, 25, 26, 34, 35, 37, 41, 42, 44, 48]
9	171	[10, 28, 19, 52, 27, 24, 6, 30, 23]	18	[0, 7, 9, 10, 12, 17, 19, 20, 21, 25, 26, 34, 35, 37, 41, 42, 44, 48]
10	194	[28, 22, 23, 26, 35, 24, 30, 10, 6, 33]	19	[0, 1, 7, 9, 10, 12, 17, 19, 20, 21, 25, 26, 34, 35, 37, 41, 42, 44, 48]
11	214	[23, 52, 22, 35, 30, 24, 6, 27, 33, 28, 10]	19	[0, 1, 7, 9, 10, 12, 17, 19, 20, 21, 25, 26, 34, 35, 37, 41, 42, 44, 48]

Table 5.2: Optimal number of clusters for the 100043-event and 20035-event Scanner case study

100043-event			200035-event		
K	#Events	Killed mutants	K	#Events	Killed mutants
16	424	**20**	15	234	**10**
18	467	**22**	30	443	**18**
22	551	**22**	60	912	**21**
30	709	**22**	73	1111	**23**

5.5 Results

Here we explain the test reduction results on the Scanette case study. We applied our approach to three different test suites introduced in Section 5.4. The Word2vec model creates a vocabulary for each of them. The number of word2vec vocab for 1026-event is 15: this corresponds to the number of different triplets. It has 1026 events from 61 shopping sessions. The entire test suite can kill 19 mutants and the goal was to reduce the number of events while maintaining the same fault detection capability viz. killing the same number of mutants. The second file, 100043-event, has 100043 events from 7079 shopping sessions. The number of word2vec vocab for this file is 18 vocab. It can kill 22 mutants. And the third one, 200035-event has 200035 events from 14443 shopping sessions that can kill 23 mutants. The number of word2vec vocab for this file is 20 vocab. These numbers are summarized in Figure 5.3 on the first row.
The sessions were vectorized and their word2vec models were constructed. After applying t-SNE and averaging sessions, we employed Elbow method to find the optimal number of clusters. The Elbow method curve of each usage log is shown in Figure 5.3 in the second row. These figures also provide two samples of clustering for two different numbers of clusters (K) around the proposed range by the Elbow method. We can observe that for the 1026-event, the sessions are distributed in some distinct clusters. As the number of events and sessions increases, the projected model tends to make a circle with some singular points in the middle. Specifically for the 200035-event for $K = 60$, in the middle of the circle, there were some sessions (points) that conveyed different clients' behavior which is to say that their series of actions were rare in comparison to the other sessions around the circle.

To observe the effect of the optimal number of clustering, we examined the K-means clustering from 2 to 10 clusters for the 1026-event, and from each cluster, the longest session was chosen. The selected sessions were executed again to see how many mutants they kill all together. Table 5.1 conveys these results. It can be seen that when $K = 10$, 10 sessions chosen from 10 clusters can

Table 5.3: The effectiveness of using t-SNE method for the 200035-event test suite

Without t-SNE

Number of events	K	Killed mutants	MutantID
236	15	9	[1, 7, 12, 17, 20, 21, 25, 26, 42]
470	30	11	[1, 7, 12, 17, 19, 20, 21, 25, 26, 42, 48]
901	60	19	[0, 1, 7, 9, 10, 12, 17, 19, 20, 21, 25, 26, 34, 35, 37, 41, 42, 44, 48]
998	70	19	[0, 1, 7, 9, 10, 12, 17, 19, 20, 21, 25, 26, 34, 35, 37, 41, 42, 44, 48]

With t-SNE

Number of events	K	Killed mutants	MutantID
234	15	10	[1, 7, 12, 17, 20, 21, 25, 26, 42, 48]
443	30	18	[0, 1, 7, 9, 10, 12, 17, 19, 20, 21, 25, 26, 34, 37, 42, 43, 45, 48]
912	60	21	[0, 1, 7, 9, 10, 12, 17, 19, 20, 21, 25, 26, 34, 35, 37, 41, 42, 43, 44, 45, 48]
1111	70	22	[0, 1, 7, 9, 10, 12, 17, 19, 20, 21, 25, 26, 34, 35, 37, 38, 41, 42, 43, 44, 45, 48]

kill all 19 mutants (highlighted by green color) that the original 1026-event test suite can kill. In this case, 194 client actions are enough and the remaining 832 actions (1026-194) can be removed. By a simple ratio, the amount of reduction is %81. For $K < 5$, the number of killed mutants is unstable and less than 19. We have shown them in yellow and pink colors. This table also provides more details on the number of selected sessions and the number of killed mutants.
The same results for two other test suites, namely 100043-event and 200035-event are presented in Table 5.2. For the 100043-event, only 467 events from 18 client sessions are enough to kill all 22 mutants. Therefore, the proposed TSR method succeeded in removing more than 99% of the redundant traces. For the 200035-event, 1111 events from 73 sessions are enough to have the same fault detection effect. Again, the same success rate is achieved.
Finally, the effectiveness of using the t-SNE method to reduce dimensions can be observed in table 5.3 which shows the number of killed mutants with and without using t-SNE for the 200035-event. With the same number of clusters (K), in all cases, using t-SNE effectively improves the number of killed mutants. This comes from the fact that t-SNE preserves and normalizes the features of each dimension when it merges them together.
From the experiments, we can conclude that treating user actions like words in sentences and building a Word2Vec+t-SNE model from them can effectively preserve users' behavior and extract their features. Applying the clustering method can group similar user sessions and in turn enables us to remove redundant sessions, which was the goal of our case study.

To have a comparison with K-means, we used Spectral clustering to construct Universal Clusters and compare the results to the K-means clustering method. The outcomes of spectral clustering frequently outperformed the K-means methods. We utilized the Python Scikit-learn library. We must basically modify the parameters of the similarity matrix and the number of cluster categories for spectral clustering. As input parameters for the Spectral Clustering function in this library, we used 'nearest neighbors' and 'precomputed' alternatively. When constructing the affinity matrix using the nearest neighbors method, we selected ten neighbors and provided the list of our data as input parameters to the function. Since we chose a custom similarity matrix for the precomputed

Table 5.4: Spectral clustering compared with K-means

Spectral clustering with t-SNE on 100043-event data			K-means clustering with t-SNE on 10043-event data	
k	#event	#Kill mutants	#Kill mutants	#event
8	171	21	20	230
9	183	21	21	261
10	207	22	22	281
11	205	21	22	283
12	218	21	22	239
13	256	22	22	351
14	263	22	22	383
15	288	22	22	429
16	299	21	22	389
17	349	22	22	444
18	352	22	22	472
19	403	22	22	448
20	389	22	22	460
30	547	23	22	677
40	891	23	22	769

method, we must configure the matrix ourselves and pass it as an input parameter to the function. In this step, we use the word2vec model for generating the affinity matrix. Table 5.4 summarizes the results of this comparison. It depicts that when both clustering methods kill an equal number of mutants, it is evident that the number of events selected with the spectral method is less than with the k-means method. However, it is essential to note that spectral clustering has the disadvantage of requiring a large amount of memory to analyze the affinity matrix.

5.6 Conclusion and Future Work

Regression testing is an essential quality-control technique during the software's maintenance phase. It is an essential activity, but large test suites can make it costly. Regression testing is accelerated by identifying and eliminating redundant tests. In this chapter, a new method to reduce regression test suites was presented.

The preliminary experiment on a simple case study shows that using a technique from NLP, namely Word2Vec, seems to provide a valuable tool for analyzing the similarity between tests in software testing contexts. As we work on traces, it also shows a potential for reducing the information to analyze lengthy software logs.

5.6.1 Summary of findings on our case study

- Word2Vec can yield meaningful feature sets for software log events.
- Using an average of the vectors of the words in a sequence associates a relevant measure for clustering software sessions made of sequences of events.
- t-SNE improves clustering and the quality of clusters of tests.
- Quite a significant test selection from random usage logs can be achieved by clustering with

Word2Vec associated with t-SNE and the Elbow method, with no loss of fault detection capabilities.

5.6.2 Threats to validity

There are obvious limits to generalizing those preliminary results.

- We just address a single case study, that is not too complex.
- We consider only test traces from random testing. Random testing is indeed an important approach and is often considered a touchstone in software testing, but we plan to consider other sources of tests, such as carefully handcrafted tests, conformance tests, and tests from DevOps approaches, etc.
- More parameters of the method should be investigated, such as the selection of representatives from each cluster, the influence of the initial abstraction of events, etc.

5.6.3 Future Work

Session averaging can be replaced by a more insightful method that can preserve the order of the events in a session. We plan to replace session averaging with another sentence embedding method like Paragraph Vectors [141] to achieve a representative vector from each session. Therefore, regarding the TSR goal of this chapter, we expect to see better results by considering the order of events.

Another attempt to replace session averaging with concept space creation to obtain vectorized session representation is also missing, which may yield a higher degree of test reduction.

We have also started investigating another direction for further reducing a test suite. Notice that we selected the longest test (session) from each UC. However, it is often the case that not all events of such a test are necessary to achieve fault detection. Therefore, we may take other criteria to select the representative client from each UC. For instance, we can select the closest client to the center of the UC, the shortest session, or a session with the most diverse events.

In the same direction, it might be possible even to reduce the selected representative sessions. There can be many redundant events that could be removed to keep the core fault-triggering capabilities of the test. We are developing an analysis of the relation between events that can trigger a fault and the necessary enablers that precede them.

Chapter 6

General Conclusions and Future Directions

This chapter provides an overview of the whole work done, including the achieved results and the challenges encountered throughout this study. In addition, based on this thesis study, we suggest numerous intriguing future directions. Through the various experiments presented in the last two chapters, we found the answer to the problem definition in the beginning of this thesis. The findings show that the suggested approach is capable of mining logs. To the best of our knowledge, the development of the proposed method, with all of its functionalities and conditions, is novel in software testing.

6.1 Conclusion and Results

Aligned with the objectives of the Philae project, we developed a methodology to create an AI model from software log files by using ML tightened with a chain of pre-processing, and conceptual space construction to create vectors to represent each sequence and universal cluster (UC) creation. The universal clusters (UCs) are essential to the verdict on sentences' semantics and their behavior for log mining tasks. The created conceptual model, and specifically the UCs reveal the causality relationship among the recorded events that was hidden from the conventional log analysis methods. The outputs and the created models can be used to perform log minimization, Software failure Prediction, Root Cause Analysis(RCA), Log Anomaly Detection, and User Behavior clustering. For specifics on some of these tasks, Software failure Prediction aims to generate proactive early warnings to avoid failures, which frequently lead to unrecoverable states. The traditional approaches to failure management are mostly passive, which deal with it after the occurrence,

while failure prediction aims to predict the failure before it happens. Moreover, as in large-scale distributed systems, it is critical to effectively analyze the root causes of incidents in order to maintain high system availability, and developers put significant effort into determining the root cause of system failures. Similarly, User Behavior clustering could be extremely useful for tasks such as task automation, which attempts to identify and automate repetitive user actions, and usability engineering, which studies how software is used and how it can be improved.

To evaluate the proposed method, we implemented the entire chain in three case studies. The first two case studies, namely, Train Ticket and Telecom, were long log files of fast-paced incoming events as well as system status monitoring, recorded in the meantime but at a slow-paced interval. In such applications, faults will manifest themselves during periods of abnormal behavior. The proposed methods detected these periods (Bug-Zones), among which, some were linked to system failures. The detected Bug-Zones cover at least 70% of the systems reboots (failures) in the Telecom case study; the 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 UC 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 an RF or UC predictor. In addition, by extracting only the pre-Bug-Zones input events, we reduced the test event set to a minimal subset of events. This subset can help software developers focus only on the bug-prone moments of software operation, reduce the cost of log analysis and root-cause detection, and finally, apply bug fixes.

The third case study, Scanette (Scanner) was a self-service shopping device with some artificially injected faults. A large usage log was available to trigger the injected faults. The goal of this case study was to reduce the large log to a small set while still triggering similar faults. The aim was to reduce debugging time and effort by intelligently selecting a tiny subset of user traces that represent the entire log. The methodology was customized to pre-process the Scanette log files and create the required models. With the aim of log reduction, the proposed method selected a tiny set (0.55% of the log file) of user traces with the same fault-triggering capability as the entire log. The role of dimensionality reduction and spectral clustering in the internal steps was evaluated and reported.

This thesis creates an ML-based log analysis methodology to abstract log information for ML techniques, create intelligent models, and prepare software artifacts to perform log mining tasks such as log minimization, test suite reduction, log anomaly detection and prediction, and finally, root cause detection. The achievements of this thesis could be used by software developers to significantly save the time and effort spent debugging faults and finding their root causes, and by system, administrators to predict system failures; although there are some aspects that are still to be explored.

6.1.1 Problems

One of the biggest problems faced in this work was in the Telecom case study. Livebox has a multitasking operating system to process parallel network operations (e.g., TV streaming, Web navigation, etc.). Hence, many times, the recorded status was not projecting the recorded incoming test events. In other words, the noisy status data was overshadowing the effect of the test execution. Some manual interventions (e.g., reboots by operators) added more ambiguity to our interpretations of the root causes. While a cluster of data confirmed a consistent causality effect of the root-cause set, another part of the data was neutral and showed no causality relation. That was

the reason why we chose another case study (Ticket Train), over which we had full control, to know the true cause of the bugs.

The main threat to the validity of this case study was that, due to human resources issues, we were not able to get complete feedback from the test team in order to evaluate the significance of the detected root-cause test events.

To evaluate the accuracy of the Bug-Zone predictor tool, it must be utilized in a real-world operational setting. Due to the lack of access to the Livebox system, we could not evaluate the predictor in action, and our evaluation was only based on the statistics of the available test data.

In the Scanner case study, we consider only test traces from random testing. Random testing is indeed an important approach and often considered a touchstone in software testing, but we plan to consider also other sources of tests, such as carefully handcrafted tests, conformance tests, tests from DevOps approaches, etc.

Another problem in this thesis is that the proposed approach has only been validated in three case studies, with different contexts and scales. Obviously, it should be assessed in 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.

6.1.2 Recommendations For Future Work

The following are possible future directions for research that can be extended from this study.

- For the anomaly detection and prediction tasks, we simply distinguish Pre-Bug-Zones from the normal operation. It would be valuable to distinguish different Bug-zone types based on their semantics. This will give an in-depth analysis of different causality effects or predict more precisely the consequence of each Bug-Zone type (e.g., system slowdown, reboot, etc.).

- There are many studies that apply various techniques and algorithms for new test generation such as [192]. We plan to extend our study to have a wider log mining functionality, for instance, apply the proposed method to generate new test cases to automate regression testing. The purpose is to discover new bugs in the system by exploring unprecedented User behavior analysis (UBA) was not within Philae's objectives and consequently, did not receive attention in this thesis. But UBA is still one of the aspects of the proposed methodology that can be applied. The UCs are directly clustering the users' behavior semantics in the Scanner case study.

- In addition, in log minimization, we need to work and get more results in the test selection part, for instance, we have to study the criteria for the selection of representatives from each cluster, instead of selecting the longest session, we can select the center member or smallest one.

- It would also be interesting to extend our approach using learning to rank techniques with partially labeled multi-modal data [193], particularly in the context of test case prioritization and automated test suite optimization; as it is often challenging to determine the order in which test cases should be executed to maximize the likelihood of detecting bugs early. Learning to rank algorithms can learn from historical data, including information about test case outcomes and code changes, to rank the test cases based on their potential to reveal defects. This prioritization can lead to more efficient testing by identifying critical test cases that are likely to uncover issues early in the testing process. Also, as software systems evolve in time, the test suite can become large and redundant, leading to increased testing time and

maintenance efforts. Learning to rank methods can be employed to automatically identify and eliminate redundant or ineffective test cases from the test suite.

Bibliography

Bibliography

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

[2] K Sri Kavya and Dr Y Prasanth. “An ensemble deepboost classifier for software defect prediction”. In: *International Journal of Advanced Trends in Computer Science and Engineering* 9.2 (2020), pp. 2021–2028.

[3] Haonan Tong, Bin Liu, and Shihai Wang. “Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning”. In: *Information and Software Technology* 96 (2018), pp. 94–111.

[4] Steffen Herbold, Alexander Trautsch, and Jens Grabowski. “A comparative study to benchmark cross-project defect prediction approaches”. In: *Proceedings of the 40th International Conference on Software Engineering*. 2018, pp. 1063–1063.

[5] Maurice Herlihy. “A methodology for implementing highly concurrent data structures”. In: *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*. 1990, pp. 197–206.

[6] Wil MP Van der Aalst, Martin Bichler, and Armin Heinzl. *Robotic process automation*. 2018.

[7] Jakob Nielsen. *Usability engineering*. Morgan Kaufmann, 1994.

[8] Vladimir A Rubin et al. “Process mining can be applied to software too!” In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 2014, pp. 1–8.

[9] Alejandro G. Martıén et al. “A survey for user behavior analysis based on machine learning techniques: current models and applications”. In: *Applied Intelligence* 51.8 (2021), pp. 6029–6055.

[10] Bahareh Afshinpour et al. “Reducing Regression Test Suites using the Word2Vec Natural Language Processing Tool.” In: *SEED/NLPaSE@ APSEC*. 2020, pp. 43–53.

[11] 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.

[12] Bahareh Afshinpour, Roland Groz, and Massih-Reza Amini. “Telemetry-based Software Failure Prediction by Concept-space Model Creation”. In: *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. IEEE. 2022, pp. 199–208.

[13] Xiang Zhou et al. “Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study”. In: *IEEE Transactions on Software Engineering* 47.2 (2018), pp. 243–260.

[14] *Microservice System Benchmark, Trainticket*. https://github.com/FudanSELab/train-ticket/. 2018.

[15] Adetokunbo AO Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. "Clustering event logs using iterative partitioning". In: *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2009, pp. 1255–1264.

[16] Canhua Wu et al. "Identifying root-cause metrics for incident diagnosis in online service systems". In: *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2021, pp. 91–102.

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

[18] Amrit Pal and Manish Kumar. "DLME: distributed log mining using ensemble learning for fault prediction". In: *IEEE Systems Journal* 13.4 (2019), pp. 3639–3650.

[19] 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)*. IEEE. 2019, pp. 140–151.

[20] Chen Luo et al. "Correlating events with time series for incident diagnosis". In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2014, pp. 1583–1592.

[21] Min Du et al. "Deeplog: Anomaly detection and diagnosis from system logs through deep learning". In: *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2017, pp. 1285–1298.

[22] Qiang Fu et al. "Where do developers log? an empirical study on logging practices in industry". In: *Companion Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 24–33.

[23] Wei Yuan et al. "How are distributed bugs diagnosed and fixed through system logs?" In: *Information and Software Technology* 119 (2020), p. 106234.

[24] Mehran Hassani et al. "Studying and detecting log-related issues". In: *Empirical Software Engineering* 23 (2018), pp. 3248–3280.

[25] Sourabh Jain et al. "Extracting the textual and temporal structure of supercomputing logs". In: *2009 International Conference on High Performance Computing (HiPC)*. IEEE. 2009, pp. 254–263.

[26] Jayanta Basak and PC Nagesh. "A user-friendly log viewer for storage systems". In: *ACM Transactions on Storage (TOS)* 12.3 (2016), pp. 1–18.

[27] Liang Huang et al. "Symptom-based problem determination using log data abstraction". In: *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research*. 2010, pp. 313–326.

[28] Adam Oliner and Jon Stearley. "What supercomputers say: A study of five system logs". In: *37th annual IEEE/IFIP international conference on dependable systems and networks (DSN'07)*. IEEE. 2007, pp. 575–584.

[29] Diana El-Masri et al. "A systematic literature review on automated log abstraction techniques". In: *Information and Software Technology* 122 (2020), p. 106276.

[30] Nicolas Aussel, Yohan Petetin, and Sophie Chabridon. “Improving performances of log mining for anomaly prediction through nlp-based log parsing”. In: *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE. 2018, pp. 237–243.

[31] Flora Amato et al. “Detect and correlate information system events through verbose logging messages analysis”. In: *Computing* 101 (2019), pp. 819–830.

[32] Byungchul Tak, Seorin Park, and Prabhakar Kudva. “Priolog: Mining important logs via temporal analysis and prioritization”. In: *Sustainability* 11.22 (2019), p. 6306.

[33] Satoru Kobayashi et al. “Mining causality of network events in log data”. In: *IEEE Transactions on Network and Service Management* 15.1 (2017), pp. 53–67.

[34] Zongze Li et al. “Converting unstructured system logs into structured event list for anomaly detection”. In: *Proceedings of the 13th International Conference on Availability, Reliability and Security*. 2018, pp. 1–10.

[35] Zongze Li et al. “Event block identification and analysis for effective anomaly detection to build reliable HPC systems”. In: *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE. 2018, pp. 781–788.

[36] Michele Pettinato et al. “Log mining to re-construct system behavior: An exploratory study on a large telescope system”. In: *Information and Software Technology* 114 (2019), pp. 121–136.

[37] Niyazi Sorkunlu, Duc Thanh Anh Luong, and Varun Chandola. “dynamicmf: A matrix factorization approach to monitor resource usage in high performance computing systems”. In: *2018 IEEE International Conference on Big Data (Big Data)*. IEEE. 2018, pp. 1302–1307.

[38] Kamal Dua et al. “CGI based syslog management system for virtual machines”. In: *Spatial Information Research* 28 (2020), pp. 475–486.

[39] Shilin He et al. “A survey on automated log analysis for reliability engineering”. In: *ACM computing surveys (CSUR)* 54.6 (2021), pp. 1–37.

[40] Paul Beynon-Davies. *Business information systems*. Bloomsbury Publishing, 2019.

[41] Haotian Wu and Qi Wang. “GEAE: Gated Enhanced Autoencoder based Feature Extraction and Clustering for Customer Segmentation”. In: *2022 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2022, pp. 1–8.

[42] Wil MP Van Der Aalst. “Process mining: discovering and improving Spaghetti and Lasagna processes”. In: *2011 IEEE symposium on computational intelligence and data mining (CIDM)*. IEEE. 2011, pp. 1–7.

[43] Gabriel M Veiga and Diogo R Ferreira. “Understanding spaghetti models with sequence clustering for ProM”. In: *Business Process Management Workshops: BPM 2009 International Workshops, Ulm, Germany, September 7, 2009. Revised Papers 7*. Springer. 2010, pp. 92–103.

[44] Ziming Zheng et al. “A practical failure prediction with location and lead time for blue gene/p”. In: *2010 International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE. 2010, pp. 15–22.

[45] Golnazsadat Zargarian et al. “Mining Patterns in Mobile Network Logs”. In: *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE. 2019, pp. 1–6.

[46] Ana Gainaru et al. “Adaptive event prediction strategy with dynamic time window for large-scale hpc systems”. In: *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*. 2011, pp. 1–8.

[47] Xiaoyu Fu et al. “Logmaster: Mining event correlations in logs of large-scale cluster systems”. In: *2012 IEEE 31st Symposium on Reliable Distributed Systems*. IEEE. 2012, pp. 71–80.

[48] Ana Gainaru et al. “Failure prediction for HPC systems and applications: Current situation and open issues”. In: *The International journal of high performance computing applications* 27.3 (2013), pp. 273–282.

[49] Ana Gainaru et al. “Fault prediction under the microscope: A closer look into HPC systems”. In: *SC’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE. 2012, pp. 1–11.

[50] Hamid Saadatfar, Hamid Fadishei, and Hossein Deldari. “Predicting job failures in AuverGrid based on workload log analysis”. In: *New Generation Computing* 30 (2012), pp. 73–94.

[51] Teerat Pitakrat. *Architecture-aware online failure prediction for software systems*. BoD–Books on Demand, 2018.

[52] Mostafa Farshchi et al. “Experience report: Anomaly detection of cloud application operations using log and cloud metric correlation analysis”. In: *2015 IEEE 26th international symposium on software reliability engineering (ISSRE)*. IEEE. 2015, pp. 24–34.

[53] Ajay Rawat et al. “A new approach for vm failure prediction using stochastic model in cloud”. In: *IETE Journal of research* 67.2 (2021), pp. 165–172.

[54] Wucherl Yoo, Alex Sim, and Kesheng Wu. “Machine learning based job status prediction in scientific clusters”. In: *2016 sai computing conference (sai)*. IEEE. 2016, pp. 44–53.

[55] Shili Xiang, Dong Huang, and Xiaoli Li. “A generalized predictive framework for data driven prognostics and diagnostics using machine logs”. In: *TENCON 2018-2018 IEEE region 10 conference*. IEEE. 2018, pp. 0695–0700.

[56] J Wang et al. “Predictive maintenance based on event-log analysis: A case study”. In: *IBM Journal of Research and Development* 61.1 (2017), pp. 11–121.

[57] Xiang Zhou et al. “Latent error prediction and fault localization for microservice applications by learning from system trace logs”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 683–694.

[58] Jing Shen et al. “Random-forest-based failure prediction for hard disk drives”. In: *International Journal of Distributed Sensor Networks* 14.11 (2018), p. 1550147718806480.

[59] Yangguang Li et al. “Predicting node failures in an ultra-large-scale cloud computing platform: an aiops solution”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29.2 (2020), pp. 1–24.

[60] Shi Jin et al. “Anomaly-detection-based failure prediction in a core router system”. In: *Proc. International Conference on Advances in System Testing and Validation Lifecycle (VALID)*. 2016.

[61] Minglu Zhao et al. “Failure prediction in datacenters using unsupervised multimodal anomaly detection”. In: *2020 IEEE International Conference on Big Data (Big Data)*. IEEE. 2020, pp. 3545–3549.

[62] Teerat Pitakrat et al. “A framework for system event classification and prediction by means of machine learning”. In: *EAI Endorsed Transactions on Self-Adaptive Systems* 1.3 (2015).

[63] Tatsuaki Kimura et al. “Proactive failure detection learning generation patterns of large-scale network logs”. In: *IEICE Transactions on Communications* 102.2 (2019), pp. 306–316.

[64] Xin Gao et al. “Incremental prediction model of disk failures based on the density metric of edge samples”. In: *IEEE Access* 7 (2019), pp. 114285–114296.

[65] Ke Zhang et al. “Automated IT system failure prediction: A deep learning approach”. In: *2016 IEEE International Conference on Big Data (Big Data)*. IEEE. 2016, pp. 1291–1300.

[66] Weibin Meng et al. “LogAnomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs.” In: *IJCAI*. Vol. 19. 7. 2019, pp. 4739–4745.

[67] Anwesha Das et al. “Desh: deep learning for system health prediction of lead times to failure in hpc”. In: *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. 2018, pp. 40–51.

[68] Shaoyu Dou, Kai Yang, and H Vincent Poor. “PC 2 A: predicting collective contextual anomalies via LSTM with deep generative model”. In: *IEEE Internet of Things Journal* 6.6 (2019), pp. 9645–9655.

[69] Rui Ren et al. “Deep convolutional neural networks for log event classification on distributed cluster systems”. In: *2018 IEEE International Conference on Big Data (Big Data)*. IEEE. 2018, pp. 1639–1646.

[70] Pin Wu et al. “Bigdata logs analysis based on seq2seq networks for cognitive Internet of Things”. In: *Future Generation Computer Systems* 90 (2019), pp. 477–488.

[71] Yijun Lin and Yao-Yi Chiang. “A Semi-Supervised Approach for Abnormal Event Prediction on Large Operational Network Time-Series Data”. In: *arXiv preprint arXiv:2110.07660* (2021).

[72] Marwa A Elsayed and Mohammad Zulkernine. “PredictDeep: security analytics as a service for anomaly detection and prediction”. In: *IEEE Access* 8 (2020), pp. 45184–45197.

[73] N Choudhary and P Singh. “Cloud computing and big data analytics”. In: *International Journal of Engineering Research and Technology* 2.12 (2013), pp. 2700–2704.

[74] Burcu Ozcelik and Cemal Yilmaz. “Seer: a lightweight online failure prediction approach”. In: *IEEE Transactions on Software Engineering* 42.1 (2015), pp. 26–46.

[75] Iago C Chaves et al. “Hard disk drive failure prediction method based on a Bayesian network”. In: *2018 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2018, pp. 1–7.

[76] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. “Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices”. In: *IEEE access* 5 (2017), pp. 3909–3943.

[77] Sandeep Dalal and Rajender Singh Chhillar. “Empirical study of root cause analysis of software failure”. In: *ACM SIGSOFT Software Engineering Notes* 38.4 (2013), pp. 1–7.

[78] Hanzhang Wang et al. “Groot: An event-graph-based approach for root cause analysis in industrial settings”. In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2021, pp. 419–429.

[79] Edward Chuah et al. “Diagnosing the root-causes of failures from cluster log files”. In: *2010 International Conference on High Performance Computing*. IEEE. 2010, pp. 1–10.

[80] Julen Kahles et al. “Automating root cause analysis via machine learning in agile software testing environments”. In: *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE. 2019, pp. 379–390.

[81] Lingzhi Wang et al. “Root-cause metric location for microservice systems via log anomaly detection”. In: *2020 IEEE International Conference on Web Services (ICWS)*. IEEE. 2020, pp. 142–150.

[82] Strategic Planning. “The economic impacts of inadequate infrastructure for software testing”. In: *National Institute of Standards and Technology* 1 (2002).

[83] Jing Xu. “Rule-based automatic software performance diagnosis and improvement”. In: *Proceedings of the 7th international workshop on Software and performance*. 2008, pp. 1–12.

[84] Karthik Nagaraj, Charles Edwin Killian, and Jennifer Neville. “Structured comparative analysis of systems logs to diagnose performance problems.” In: *NSDI*. 1. 2012, p. 353.

[85] He Yan et al. “G-rca: a generic root cause analysis platform for service quality management in large ip networks”. In: *Proceedings of the 6th International COnference*. 2010, pp. 1–12.

[86] Mike Chen et al. “Failure diagnosis using decision trees”. In: *International Conference on Autonomic Computing, 2004. Proceedings.* IEEE. 2004, pp. 36–43.

[87] Vicent Rodrigo et al. “Causal analysis for alarm flood reduction”. In: *IFAC-PapersOnLine* 49.7 (2016), pp. 723–728.

[88] Selçuk Emre Solmaz et al. “ALACA: A platform for dynamic alarm collection and alert notification in network management systems”. In: *International Journal of Network Management* 27.4 (2017), e1980.

[89] Richard Jarry, Satoru Kobayashi, and Kensuke Fukuda. “A quantitative causal analysis for network log data”. In: *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE. 2021, pp. 1437–1442.

[90] Cody Watson et al. “A systematic literature review on the use of deep learning in software engineering research”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31.2 (2022), pp. 1–58.

[91] Jussi Roberts. “Iterative Root Cause Analysis Using Data Mining in Software Testing Processes”. PhD thesis. Master’s Thesis, Degree Programme in Information Processing Science, 2016.

[92] Qingchao Shen et al. “A comprehensive study of deep learning compiler bugs”. In: *Proceedings of the 29th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 2021, pp. 968–980.

[93] BR Grishma and C Anjali. “Software root cause prediction using clustering techniques: A review”. In: *2015 Global Conference on Communication Technologies (GCCT)*. IEEE. 2015, pp. 511–515.

[94] Ru Zhang et al. “An empirical study on program failures of deep learning jobs”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020, pp. 1159–1170.

[95] Peter Jackson. “Introduction to expert systems”. In: (1986).

[96] Risto Vaarandi. “A data clustering algorithm for mining patterns from event logs”. In: *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003)(IEEE Cat. No. 03EX764)*. Ieee. 2003, pp. 119–126.

[97] J Christopher Westland. “The cost of errors in software development: evidence from industry”. In: *Journal of Systems and Software* 62.1 (2002), pp. 1–9.

[98] 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)*. IEEE. 2020, pp. 176–183.

[99] Wei Xu et al. “Detecting large-scale system problems by mining console logs”. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009, pp. 117–132.

[100] Qingwei Lin et al. “Log clustering based problem identification for online service systems”. In: *Proceedings of the 38th International Conference on Software Engineering Companion*. 2016, pp. 102–111.

[101] Shilin He et al. “Identifying impactful service system problems via log analysis”. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2018, pp. 60–70.

[102] Yinglung Liang et al. “Failure prediction in ibm bluegene/l event logs”. In: *Seventh IEEE International Conference on Data Mining (ICDM 2007)*. IEEE. 2007, pp. 583–588.

[103] Wei Xu et al. “Online system problem detection by mining patterns of console logs”. In: *2009 ninth IEEE international conference on data mining*. IEEE. 2009, pp. 588–597.

[104] Jian-Guang Lou et al. “Mining Invariants from Console Logs for System Problem Detection.” In: *USENIX annual technical conference*. 2010, pp. 1–14.

[105] Animesh Nandi et al. “Anomaly detection using program control flow graph mining from execution logs”. In: *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 2016, pp. 215–224.

[106] Jian-Guang Lou et al. “Mining program workflow from interleaved traces”. In: *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2010, pp. 613–622.

[107] Tong Jia et al. “LogFlash: Real-time Streaming Anomaly Detection and Diagnosis from System Logs for Large-scale Software Systems”. In: *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2021, pp. 80–90.

[108] Kenji Yamanishi and Yuko Maruyama. “Dynamic syslog mining for network failure monitoring”. In: *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. 2005, pp. 499–508.

[109] Shilin He et al. “Experience report: System log analysis for anomaly detection”. In: *2016 IEEE 27th international symposium on software reliability engineering (ISSRE)*. IEEE. 2016, pp. 207–218.

[110] Marcello Cinque, Raffaele Della Corte, and Antonio Pecchia. “Micro2vec: Anomaly detection in microservices systems by mining numeric representations of computer logs”. In: *Journal of Network and Computer Applications* 208 (2022), p. 103515.

[111] Xu Zhang et al. “Robust log-based anomaly detection on unstable log data”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 807–817.

[112] Bin Xia et al. “Loggan: a log-level generative adversarial network for anomaly detection using permutation event modeling”. In: *Information Systems Frontiers* 23 (2021), pp. 285–298.

[113] Siyang Lu et al. “Detecting anomaly in big data system logs using convolutional neural network”. In: *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*. IEEE. 2018, pp. 151–158.

[114] Fucheng Liu et al. “Log2vec: A heterogeneous graph embedding based approach for detecting cyber threats within enterprise”. In: *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 2019, pp. 1777–1794.

[115] Xiaoyun Li et al. “Swisslog: Robust and unified deep learning based log anomaly detection for diverse faults”. In: *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2020, pp. 92–103.

[116] Jin Wang et al. “LogEvent2vec: LogEvent-to-vector based anomaly detection for large-scale logs in internet of things”. In: *Sensors* 20.9 (2020), p. 2451.

[117] Haixuan Guo, Shuhan Yuan, and Xintao Wu. “Logbert: Log anomaly detection via bert”. In: *2021 international joint conference on neural networks (IJCNN)*. IEEE. 2021, pp. 1–8.

[118] Lin Yang et al. “Semi-supervised log-based anomaly detection via probabilistic label estimation”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 1448–1460.

[119] Min Li et al. “MDFULog: Multi-Feature Deep Fusion of Unstable Log Anomaly Detection Model”. In: *Applied Sciences* 13.4 (2023), p. 2237.

[120] Piotr Ryciak, Katarzyna Wasielewska, and Artur Janicki. “Anomaly detection in log files using selected natural language processing methods”. In: *Applied Sciences* 12.10 (2022), p. 5089.

[121] Chunkai Zhang et al. “LayerLog: Log sequence anomaly detection based on hierarchical semantics”. In: *Applied Soft Computing* 132 (2023), p. 109860.

[122] Thorsten Wittkopp et al. “PULL: Reactive Log Anomaly Detection Based On Iterative PU Learning”. In: *arXiv preprint arXiv:2301.10681* (2023).

[123] Jieming Zhu et al. “Tools and benchmarks for automated log parsing”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE. 2019, pp. 121–130.

[124] Rui Chen et al. “Logtransfer: Cross-system log anomaly detection for software systems with transfer learning”. In: *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2020, pp. 37–47.

[125] Ian J Goodfellow et al. “Generative adversarial networks.” In: *Commun. Acm* 63.11 (2020), pp. 139–144.

[126] Gregg Rothermel et al. “Prioritizing test cases for regression testing”. In: *IEEE Transactions on software engineering* 27.10 (2001), pp. 929–948.

[127] Mark Harman. “Making the case for MORTO: Multi objective regression test optimization”. In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE. 2011, pp. 111–114.

[128] Liang You and YanSheng Lu. “A genetic algorithm for the time-aware regression testing reduction problem”. In: *2012 8th International Conference on Natural Computation*. IEEE. 2012, pp. 596–599.

[129] Stuart Lloyd. “Least squares quantization in PCM”. In: *IEEE transactions on information theory* 28.2 (1982), pp. 129–137.

[130] J MacQueen. “Some methods for classification and analysis of multivariate observations”. In: *Proc. 5th Berkeley Symposium on Math., Stat., and Prob.* 1965, p. 281.

[131] Edward W Forgy. “Cluster analysis of multivariate data: efficiency versus interpretability of classifications.” biometrics 21”. In: (1965).

[132] Maria CV Nascimento and Andre CPLF De Carvalho. “Spectral methods for graph clustering–a survey”. In: *European Journal of Operational Research* 211.2 (2011), pp. 221–231.

[133] Leo Breiman. “Random forests”. In: *Machine learning* 45 (2001), pp. 5–32.

[134] Leo Breiman. “Bagging predictors”. In: *Machine learning* 24 (1996), pp. 123–140.

[135] Shusen Liu et al. “Visual exploration of semantic relationships in neural word embeddings”. In: *IEEE transactions on visualization and computer graphics* 24.1 (2017), pp. 553–562.

[136] Tomas Mikolov et al. “Efficient estimation of word representations in vector space”. In: *arXiv preprint arXiv:1301.3781* (2013).

[137] Jeffrey Pennington, Richard Socher, and Christopher D Manning. “Glove: Global vectors for word representation”. In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543.

[138] Tomas Mikolov et al. “Distributed representations of words and phrases and their compositionality”. In: *Advances in neural information processing systems*. 2013, pp. 3111–3119.

[139] Laurens Van der Maaten and Geoffrey Hinton. “Visualizing data using t-SNE.” In: *Journal of machine learning research* 9.11 (2008).

[140] Massih-Reza Amini. “Interactive Learning for Text Summarization”. In: *Proceedings of the PKDD/MLTIA Workshop on Machine Learning and Textual Information Access*. Lyon - France, 2000.

[141] Quoc Le and Tomas Mikolov. “Distributed representations of sentences and documents”. In: *International conference on machine learning*. 2014, pp. 1188–1196.

[142] Zellig S Harris. “Distributional structure”. In: *Word* 10.2-3 (1954), pp. 146–162.

[143] Ziyi Yang, Chenguang Zhu, and Weizhu Chen. “Parameter-free sentence embedding via orthogonal basis”. In: *arXiv preprint arXiv:1810.00438* (2018).

[144] Markus Goldstein and Seiichi Uchida. “A comparative evaluation of unsupervised anomaly detection algorithms for multivariate data”. In: *PloS one* 11.4 (2016), e0152173.

[145] Markus M Breunig et al. “LOF: identifying density-based local outliers”. In: *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 2000, pp. 93–104.

[146] Sahand Hariri, Matias Carrasco Kind, and Robert J Brunner. “Extended isolation forest”. In: *IEEE Transactions on Knowledge and Data Engineering* 33.4 (2019), pp. 1479–1489.

[147] Adam Oliner, Archana Ganapathi, and Wei Xu. “Advances and challenges in log analysis”. In: *Communications of the ACM* 55.2 (2012), pp. 55–61.

[148] Alina Oprea et al. “Detection of early-stage enterprise infection by mining large-scale log data”. In: *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE. 2015, pp. 45–56.

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

[150] Pinjia He et al. “Towards automated log parsing for large-scale log data analysis”. In: *IEEE Transactions on Dependable and Secure Computing* 15.6 (2017), pp. 931–944.

[151] Hetong Dai et al. “Logram: Efficient Log Parsing Using n n-Gram Dictionaries”. In: *IEEE Transactions on Software Engineering* 48.3 (2020), pp. 879–892.

[152] Pinjia He et al. “Drain: An online log parsing approach with fixed depth tree”. In: *2017 IEEE international conference on web services (ICWS)*. IEEE. 2017, pp. 33–40.

[153] Laurens van der Maaten and Geoffrey Hinton. “Visualizing data using t-SNE”. In: *Journal of machine learning research* 9.Nov (2008), pp. 2579–2605.

[154] Radim Rehurek and Petr Sojka. “Gensim–python framework for vector space modelling”. In: *NLP Centre, Faculty of Informatics, Masaryk University, Brno, Czech Republic* 3.2 (2011).

[155] Alexis Conneau et al. “Supervised learning of universal sentence representations from natural language inference data”. In: *arXiv preprint arXiv:1705.02364* (2017).

[156] Sanjeev Arora, Yingyu Liang, and Tengyu Ma. “A simple but tough-to-beat baseline for sentence embeddings”. In: (2016).

[157] Jean-François Pessiot et al. “Improving document clustering in a learned concept space”. In: *Information processing & management* 46.2 (2010), pp. 180–192.

[158] Zena M Hira and Duncan F Gillies. “A review of feature selection and feature extraction methods applied on microarray data”. In: *Advances in bioinformatics* 2015 (2015).

[159] Weikuan Jia et al. “Feature dimensionality reduction: a review”. In: *Complex & Intelligent Systems* 8.3 (2022), pp. 2663–2693.

[160] Rung-Ching Chen et al. “Selecting critical features for data classification based on machine learning methods”. In: *Journal of Big Data* 7.1 (2020), p. 52.

[161] Christine Dewi and Rung-Ching Chen. “Human activity recognition based on evolution of features selection and random Forest”. In: *2019 IEEE international conference on systems, man and cybernetics (SMC)*. IEEE. 2019, pp. 2496–2501.

[162] Celine Vens and Fabrizio Costa. “Random forest based feature induction”. In: *2011 IEEE 11th international conference on data mining*. IEEE. 2011, pp. 744–753.

[163] Oumaima Alaoui Ismaili, Vincent Lemaire, and Antoine Cornuéjols. “A supervised methodology to measure the variables contribution to a clustering”. In: *Neural Information Processing: 21st International Conference, ICONIP 2014, Kuching, Malaysia, November 3-6, 2014. Proceedings, Part I 21*. Springer. 2014, pp. 159–166.

[164] Y Alghofaili. *Interpretable K-Means: clusters feature importances*.

[165] Fareed Zandkarimi et al. “A generic framework for trace clustering in process mining”. In: *2020 2nd International Conference on Process Mining (ICPM)*. IEEE. 2020, pp. 177–184.

[166] Xiao Yu et al. “Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs”. In: *ACM SIGARCH Computer Architecture News* 44.2 (2016), pp. 489–502.

[167] 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*. 2020, pp. 315–326.

[168] Yong Xu et al. “Improving service availability of cloud systems by predicting disk error”. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018, pp. 481–494.

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

[170] Linchang Fan et al. “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)*. IEEE. 2021, pp. 1–5.

[171] Robert L. Thorndike. “Who belongs in the family”. In: *Psychometrika* (1953), pp. 267–276.

[172] Andrew P Bradley. “The use of the area under the ROC curve in the evaluation of machine learning algorithms”. In: *Pattern recognition* 30.7 (1997), pp. 1145–1159.

[173] David MW Powers. “Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation”. In: *arXiv preprint arXiv:2010.16061* (2020).

[174] Xiaofang Zhang, Huamao Shan, and Ju Qian. “Resource-aware test suite optimization”. In: *2009 Ninth International Conference on Quality Software*. IEEE. 2009, pp. 341–346.

[175] August Shi et al. “Balancing trade-offs in test-suite reduction”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014, pp. 246–256.

[176] Jeff Offutt, Jie Pan, and Jeffrey M Voas. “Procedures for reducing the size of coverage-based test sets”. In: *Proceedings of the 12th International Conference on Testing Computer Software*. ACM Press New York. 1995, pp. 111–123.

[177] Saif Ur Rehman Khan et al. “A systematic review on test suite reduction: Approaches, experiment’s quality evaluation, and guidelines”. In: *IEEE Access* 6 (2018), pp. 11816–11841.

[178] Frédéric Tamagnan et al. “Regression Test Generation by Usage Coverage Driven Clustering on User Traces”. In: *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2023, pp. 82–89.

[179] Carmen Coviello, Simone Romano, and Giuseppe Scanniello. “Poster: CUTER: ClUstering-based TEst suite reduction”. In: *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE. 2018, pp. 306–307.

[180] Carmen Coviello et al. “Clustering support for inadequate test suite reduction”. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2018, pp. 95–105.

[181] Emilio Cruciani et al. “Scalable approaches for test suite reduction”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 419–429.

[182] André Reichstaller et al. “Test suite reduction for self-organizing systems: a mutation-based approach”. In: *Proceedings of the 13th International Workshop on Automation of Software Test*. 2018, pp. 64–70.

[183] Hermann Felbinger, Franz Wotawa, and Mihai Nica. “Test-suite reduction does not necessarily require executing the program under test”. In: *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE. 2016, pp. 23–30.

[184] Muhammad Rashid Naeem et al. “A machine learning approach for classification of equivalent mutants”. In: *Journal of Software: Evolution and Process* 32.5 (2020), e2238.

[185] Xingmin Luo, Fan Ping, and Mei-Hwa Chen. “Clustering and tailoring user session data for testing web applications”. In: *2009 International Conference on Software Testing Verification and Validation*. IEEE. 2009, pp. 336–345.

[186] Vinicius HS Durelli et al. “Machine learning applied to software testing: A systematic mapping study”. In: *IEEE Transactions on Reliability* 68.3 (2019), pp. 1189–1212.

[187] Joanna Strug and Barbara Strug. “Machine learning approach in mutation testing”. In: *Testing Software and Systems: 24th IFIP WG 6.1 International Conference, ICTSS 2012, Aalborg, Denmark, November 19-21, 2012. Proceedings 24*. Springer. 2012, pp. 200–214.

[188] Mike Papadakis et al. “Mutation testing advances: an analysis and survey”. In: *Advances in Computers*. Vol. 112. Elsevier, 2019, pp. 275–378.

[189] Jeff Mitchell and Mirella Lapata. “Composition in distributional models of semantics”. In: *Cognitive science* 34.8 (2010), pp. 1388–1429.

[190] John Wieting et al. “Towards universal paraphrastic sentence embeddings”. In: *arXiv preprint arXiv:1511.08198* (2015).

[191] G. Linderman and S. Steinerberger. “Clustering with t-SNE, provably”. In: *SIAM J. Math. Data Sci.* 1 (2019), pp. 313–332.

[192] Mark Utting et al. “Identifying and generating missing tests using machine learning on execution traces”. In: *2020 IEEE International Conference On Artificial Intelligence Testing (AITest)*. IEEE. 2020, pp. 83–90.

[193] Nicolas Usunier, Massih-Reza Amini, and Cyril Goutte. “Multiview Semi-Supervised Learning for Ranking Multilingual Documents”. In: *Proceedings of the 2011 European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part III*. ECML PKDD’11. Athens, Greece: Springer-Verlag, 2011, pp. 443–458. ISBN: 9783642238079.