Fault Prediction and Localization with Test Logs

Anunay Amar

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science (Computer Science) at
Concordia University
Montréal, Québec, Canada

June 2018

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Anunay Amar**

Entitled: **Fault Prediction and Localization with Test Logs**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

______________________________ Chair

______________________________ Examiner
Dr. Emad Shihab

______________________________ Examiner
Dr. Tse-Hsun (Peter) Chen

______________________________ Supervisor
Dr Peter C. Rigby

Approved by ______________________________
Dr Volker Haarslev, Graduate Program Director

XX June 2018 ______________________________
Dr Amir Asif, Dean
Faculty of Engineering and Computer Science

Abstract

Fault Prediction and Localization with Test Logs

Anunay Amar

Software testing is an integral part of modern software development. However, test runs produce 1000's of lines of logged output that make it difficult to find the cause of a fault in the logs. This problem is exacerbated by environmental failures that distract from product faults. In this thesis we present techniques that reduce the number of log lines that testers manually investigate while still finding a maximal number of faults.

We observe that the location of a fault should be contained in the lines of a failing log. In contrast, a passing log should not contain the lines related to a failure. Lines that occur in both a passing and failing log introduce noise when attempting to find the fault in a failing log. We introduce a novel approach where we remove the lines that occur in the passing log from the failing log.

After removing these lines, we use information retrieval techniques to flag the most probable lines for investigation. We modify TF-IDF to identify the most relevant log lines related to past product failures. We then vectorize the logs and develop an exclusive version of KNN to identify which logs are likely to lead to product faults and which lines are the most probable indication of the failure.

Our best approach, FaultFlagger finds 89% of the total faults and flags only 0.5% of lines for inspection. FaultFlagger drastically outperforms the previous work CAM. We implemented FaultFlagger as a tool at Ericsson where it presents daily fault prediction summaries to testers.

Acknowledgments

I would like to take this opportunity to show my gratitude towards the people who have played an indispensable role in this memorable journey.

Foremost, I would like to express my sincere gratitude and respect towards my thesis supervisor, Dr. Peter Rigby. This work would not have been possible without his guidance, support and encouragement. His undying patience and guidance have helped me in all phases of this journey, from carrying out research to writing this thesis. Sincerely, I could not have asked for a better supervisor. In addition, I would also like to thank Ericsson Inc. for providing us with data and necessary hardware. My special thanks to Chris Griffiths and Gary McKenna for their valuable feedback.

I am also thankful to Dr. Emad Shihab and Dr. Weiyi Shang for their time and guidance. Additionally, I would like to thank Concordia University for providing me with an opportunity to be a part of it.

Last but not the least, I would like to thank my parents and my brother for their love and constant support. I could not have accomplish this without their support and motivation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Large complex software systems have 1000's of test runs each day leading to 10's of thousands of test log lines [51, 35, 32]. Test cases fail primarily due to two reasons during software testing: a) fault in the product code and b) issues pertaining to the test environment [78]. If a test fails due to a fault in the source code, then a bug report is created and developers are assigned to resolve the product fault. However, if a test fails due to a non-product issue, then the test is usually re-executed and often the test environment is fixed. Non-product test failures are a significant problem. For example, Google reports that 84% of tests that fail for the first time are non-product or flaky failures [51]. At Microsoft, techniques have been developed to automatically classify and ignore false test alarms [32]. At Huawei researchers have classified test failures into multiple categories including product vs environmental failure to facilitate fault identification [35].

In this work we focus on the Ericsson team that is responsible for testing cellular basestation software. The software that runs on these base stations contains not only complex signalling logic with stringent real-time constraints, but also must be highly reliable, providing safety critical services, such as 911 calling. The test environment involves specialized test hardware and RF signalling that adds additional complexity to the test environment. For example, testers need to simulate cellular devices, such as when a base station is overwhelmed by requests from cell users at a music concert.

To identify the cause of a test failure, software testers go through test execution logs and inspect the log lines. The inspection relies on a tester's experience, expertise, intuition, past run information, and regular expressions crafted using historical execution data. The process of inspection of the failed test execution log is tedious, time consuming, and makes software testing more costly [72].

Discussions with Ericsson developers revealed two challenges in the identification of faults in a failing test log: 1) the complex test environment introduces many non-product test failures and 2) the logs contain an overwhelming amount of detailed information. To solve these problems, we mine the test logs to predict which test failures will lead to product faults and which lines in those logs are most likely to reveal the cause of the fault. To assess the quality of our techniques we use two evaluation metrics on historical test log data: the number of faults found, *FaultsFound*, and the number of log lines investigated to find those faults, *LogLinesFlagged*. An overview of the four techniques are described below.

1. **CAM***: TF-IDF & KNN*

CAM was implemented at Huawei to categorized failing test logs [35]. Testers had manually classified a large sample of failing test logs into categories including product and environment failures. CAM runs TF-IDF across the logs to determine which terms had the highest importance. They create vectors and rank the logs using cosine similarity. An unseen test failure log is categorized, *e.g.,* product vs environment failure, by examining the categories of the K nearest neighbours (KNN).

Although CAM categorizes logs, it does not flag lines within a log for investigation. The logs at Ericsson contain hundreds of log lines making a simple categorization of a log as fault or product unhelpful. Our goal is to flag the smallest number of lines while identifying as many faults as possible. When we replicate CAM on Ericsson data only 47% of the faults are found. Since the approach cannot flag specific lines within a log, any log that is categorized as having a product fault, must be investigated in its entirety.

2. **SkewCAM***: CAM with* **EKNN**

Ericsson's test environment is highly complex with RF signals and specialized base-station test hardware. This environment leads to a significant proportion of environmental test failures relative to the number of product test failures. In our study we found that the data was highly skewed because of the significant proportion of environmental failures. Teams of testers exclusively analyze log test failures each day. They want to ensure that all product faults are found. We modify the standard K Nearest Neighbour (KNN) classification approach to act in an exclusive manner. With Exclusive K Nearest Neighbour (*EKNN*), instead of voting during classification, if any past log among K neighbours has been associated with a product fault, then the current log will flagged as product fault. SkewCAM, which replaces KNN with *EKNN*, finds 88% of *FaultsFound* with 28% of the log lines being flagged for investigation.

3. **LogLiner***: Line-IDF &* **EKNN**

SkewCAM accurately identifies logs that lead to product faults, but still requires the tester to examine 1/3 of the total log lines. Our goal is to flag fewer lines to provide accurate fault localization.

The unit of analysis for SkewCAM is each individual term in a log. Using our abstraction and cleaning approaches, we remove run specific information and ensure that each log line is unique. We are then able to use Inverse Document Frequency (IDF) at the line level to determine which lines are rare across all failing logs and likely to provide superior fault identification for a particular failure. LogLiner, Line-IDF & *EKNN*, can identify 84% of product faults while flagging only 3% of the log lines. There is a slight reduction in *FaultsFound* found but a near 10 fold reduction in *LogLinesFlagged* for inspection.

4.* FaultFlagger*: PastFaults* ∗ *Line-IDF & EKNN

Inverse Document Frequency (IDF) is usually weighted by Term Frequency (TF). Instead of using a generic term frequency for weight, we use the number of times a log line has been associated with a product fault in the past. The result is that lines with historical faults are weighed more highly. FaultFlagger, identifies 89% of *FaultsFound* while only flagging 0.5% of the log lines. FaultFlagger finds the same number of faults as SkewCAM, but flags less than 1% of the log lines compared to SkewCAM's 27%.

The thesis is divided into the following chapters. In Chapter 2, we conduct a background literature review that covers both broad testing topics and the specific literature on which this thesis is based. The topics include the types and purposes of testing, the challenges in software testing and research into software fault localization, statistical fault prediction, log processing and abstraction, and categorization of test failures. In Chapter 3, we provide the background on the Ericsson test process and the data that we analyze. In Section 3.1, we detail our methodology including our log abstraction, cleaning, diffing, and classification methodologies. In Section 3.2, we describe our evaluation setup. In Chapter 4, we provide the results for our four log prediction and line flagging approaches and discuss threats to validity. In section 4.5, we discuss how the best approach was implement as a tool at Ericsson. In Chapter 5, we contrast the approaches based on the number of *FaultsFound* and *LogLinesFlagged* for inspection. In Chapter 6, we position our work in the context of the existing literature, and provide a concluding discussion of our novel contributions.

Chapter 2

Background Literature Review

We break the related work into the following categories:

- Types and purposes of testing
- Challenges in software testing
- Research into software fault localization
- Research into statistical fault prediction
- Research into log processing and abstraction
- Research into categorization of test failures

2.1 Types and purpose of testing

Software testing is a technique to ensure the quality of a software product. The testing process checks whether a software product is working correctly often with respect to set standards and requirements [78]. Any deviation shown by a software product in its behaviour implies a bug and can lead to a software failure [48]. A software system can fail primarily due to two reasons, a) bug in source code or b) a bug in software environment [78].

To catch software bugs, we perform testing at different levels [78]. For example, we conduct unit-testing to test individual software components, integration testing to test a group of software components, and system testing to test the entire software product. We also use different testing strategies based on the testing requirements [48, 1, 24]. For example, we perform structural testing to test the internal structure of the software product, whereas we use requirement testing to test the external behaviour of the software product.

According to a survey carried out by Ng *et al.* [58], the most popular testing activities are test case design, documentation of test results, regression testing to test changed code, creation of test objectives, and updating test plans according to new requirements and specifications. In summary, testing is a complex task and involves many different activities.

2.2 Challenges in software testing

Software testing is a complex activity as a result there are many challenges that we face during software testing. One of the most important challenges is to improve the number of bugs found during software testing [6]. To partially solve this problem Hutchins *et al.* suggested a test selection technique that is quite effective in detecting faults and uses criteria like code coverage, control-flow, and data-flow [34]. The problem can also be ameliorated with the help of exhaustive testing. However, performing exhaustive testing on a software system is usually prohibitively expensive [44]. Consequently, Kuhn *et al.* suggested a technique to perform pseudo-exhaustive testing by exhaustively testing a small subset of parameters that causes software failures. Basili *et al.* performed a study and suggested that different factors like testing technique, software type, fault type, tester experience, and any interaction among these factors affects the efficiency of the testing system [5]. Consequently, we should take both product and process factors into account to improve the test effectiveness.

Test size and automation

Large and complex software systems further increase the complexity faced during software testing [6]. For example, it becomes quite hard to find the correct order of class integration when we perform the integration and the testing of object-oriented system [9]. Several researchers have suggested techniques to reduce the complexity faced during software testing [10, 59, 76]. Complexity involved in software testing can be ameliorated using test automation. However, completely automating the testing system has its own challenges [6]. To solve this challenge, Godefroid *et al.* suggested a technique called DART (Directed automated random testing) that can perform a fully automated unit-testing on any software product that compiles [25]. DART automatically generates the test driver that performs random testing. DART also performs dynamic analysis and generates new test inputs to achieve maximum test coverage. The technique can successfully detect standard errors like program crash, assertion violation, and non-termination etc. Another unit test automation technique called PUT (Parameterized unit tests) was proposed by Microsoft researchers Tillman *et al.* [73]. In this technique, symbolic execution is used to construct parameterized unit test cases that provides maximum code coverage.

Generating tests and mutants

An approach that theoretically identifies additional faults is mutation testing [61, 18]. Mutation testing is a fault-based technique that helps testers in creating the test cases that detect well-defined class of faults. Mutants are simple faults present in a software program. When a test case detects a mutant, the mutant is killed. Dead mutants are not executed by later test cases. To perform test case minimization, a mutation score is calculated for different sets of test cases, where mutation score denotes the ratio of total killed mutants to the total killable mutants. During test case minimization, a minimal set of test cases is selected that generates the highest mutation score. Fuzz testing builds upon mutation testing. In traditional fuzz testing, random mutations are applied to well-formed inputs and the resulting values are evaluated for faults. Whereas, in whitebox fuzz testing, new inputs are calculated with the help of old inputs and various conditional constraints faced by the old input. A whitebox fuzz testing approach was suggested by Godefroid *et al.* [26] to quickly detect security vulnerabilities. This approach calculates the new input so that we can maximize the code coverage, and test and uncover security vulnerabilities in additional code statements.

Test cost and reprioritization

Software testing is expensive and it is estimated that testing consumes more than 50 percent of the entire software development budget [6]. To reduce the cost of software testing several approaches have been suggested. Biffl *et al.* proposed a value-based software engineering approach that uses test manager's knowledge to select tests to reduce the overall cost [8]. Herzig *et al.* [31] developed a tool called THEO that uses past test execution data to determine how effective a test is in finding the bugs. Ineffective tests are skipped based on a cost function. THEO skips the test without affecting the quality of the software product by ensuring that at a predefined point all tests will be run. To reduce the cost during regression testing, Kim *et al.* [41] proposed a test prioritization technique that uses historical execution data. According to the prioritization technique, test cases that are historically better at uncovering the software faults are selected first. Each test cases are assigned a probability that decides when the test will be selected. These probabilities are calculated using a exponential weighted moving average, and the past execution performance of the test. To save cost, Elbaum *et al.* [20] applied a test selection and prioritization technique during continuous integration and development cycle. The technique uses historical execution data to perform test selection and prioritization. The technique uses a time window to track how recently test suites have been executed and revealed failures.

2.3 Research into software fault localization

Software testing helps in finding faults in a software product. Although a test failure indicates a problem in the system, the actual location of the fault can be difficult to find. Identifying the location of the fault has historically been a manual, tedious, and time consuming task. Furthermore, the success of manual fault localization relies on the software developer's experience and judgment [80]. There has been substantial research to facilitate and automate the fault localization process.

As traditional fault localization testing is not effective in finding software faults, many advanced software fault localization techniques have been suggested. Some of the popular fault localization techniques are: program slicing-based techniques, program spectrum-based techniques, statistics-based techniques, program-state based techniques, machine learning-based techniques, and information-retrieval based techniques [80].

Slice-Based Techniques

Program slicing based fault localization technique was first introduced by Weiser *et al.* [77]. It is a technique that helps in uncovering all the statements in a program that directly or indirectly affect a particular variable that produces the wrong output. Many different variants of program slicing based fault localization techniques have been suggested [45, 81, 17]. For example, Shinji *et al.* suggested a static program slicing to locate the software fault. Whereas, Franz *et al.* suggested a dynamic program slicing to locate the software faults. Dynamic program slicing, has a advantage over static program slicing as dynamic program slicing takes program execution into account while calculating the number statements that affects the variable of interest, and consequently flags fewer number of suspicious statements. To further reduce the number of suspicious statements, DeMillo *et al.* used dynamic program slicing along with mutation-based testing. Mutation-based testing is a technique that helps in determining the adequacy of a test set against a collection of mutant programs. Nevertheless, all slicing based technique suffers from a common disadvantage. The number of suspicious statements flagged by these techniques is overwhelmingly large and forces software testers to look at huge chunks of code to identify the software fault [80].

Program Spectrum-Based Techniques

Different variants of a popular ESHS (Executable Statement Hit Spectrum) based technique called Tarantula were suggested by researchers for the purpose of fault localization [13, 37, 16]. ESHS is a technique that indicates which part of the program under testing has been covered during the test execution. With the help of the above information one can identify the statements involved during a software failure. Tarantula is a ESHS based technique that calculates the suspiciousness of a program statement with the help of coverage information and execution result i.e. success or failure.

Statistics-Based Techniques

Chilimbi *et al.* suggested a statistics-based fault localization technique that uses path profiles [12]. Path profiles are collected during test execution and are assigned importance score. Finally paths with top scores are presented as a root cause of the software fault.

Program State-Based Techniques

Program state-difference-based fault localization technique was proposed by Zeller and Hildebrandt [85, 86]. In this technique, states of successful test and failed test are compared using their memory graphs. Memory graph is a data structure that represents the state of the program [87]. The Memory graph contains all the variables and values present in the program, edges of the graph denotes different operations like pointer dereferencing, variable access etc.

Machine Learning-Based Techniques

Some researchers even applied sophisticated machine learning techniques like SVM, KNN to locate software faults [69, 57, 11, 21]. For example, Neuhaus *et al.* used SVM and KNN to identify vulnerable software components. They trained their model on the patterns that frequently occur in vulnerable software components. Similarly, Brun *et al.* [11] used SVM and decision tree, and trained their models on properties of erroneous programs and fixed version of them. Using this technique they were able to identify the program properties that indicate errors.

2.4 Research into statistical fault prediction

Predicting software faults is an active research field. Most fault prediction techniques predict whether a given software module, file, or commit will contain faults. Some of the most popular and recent fault prediction techniques uses statistical regression models and machine learning models to predict faults in software modules [23, 52, 60, 2, 39, 40, 15, 43, 70, 49, 39, 50, 42]. Many of these fault prediction techniques uses similar data metrics and classification techniques to train their fault prediction model. For example, Ghotra *et al.* [23] train their model with the help of statistical techniques like Naive Bayes and simple logistic regression. The measure in these models include like Lines of Code (LOC), cyclomatic complexity, measures of behavioural and structural design properties of classes, objects, and the relations between them. Moeyersoms *et al.* used data features like LOC, McCabe [68] and Halstead [3] metrics to train their Random Forest and SVM based fault prediction technique. Similarly, Okutan *et al.* [60] used a Bayesian networks prediction model and trained it on metrics including LOC, lack of code quality (LOCQ) for source code, number of developers (NOD), coupling between objects (CBO), weighted method per class (WMC), and lack of cohesion of methods (LCOM). To predict faults in commits, Kamei *et al.* [38] suggested a statistical bug model that measures the risk associated with changing a set of files. The model uses information like the number of changes, change size, history of changes, and developer experience to measure the risk involved in a commit. Herizg [30] performed preliminary work combining the measures such as code churn, organizational structure, and pre-release defects with pre-release test failures to predict the defects at the file and Microsoft binary level.

2.5 Research into log processing and abstraction

Traditional fault localization techniques include the use of program logs, assertions, breakpoints in a debugger, and profilers [19, 63, 14, 4]. For example, logging statements are inserted into software code to monitor the program behaviour. If a program shows an unusual behaviour then a tester manually inspects the log to identify the problem. However, manually inspecting the test log is tedious and time consuming. Consequently, there is a need for better alternative to find software faults.

Test logs contain a plethora of useful information. We can use the information provided by the logs to perform debugging, detection of security threats, anomaly detection, and many other tasks [55]. However, performing log analysis on huge unstructured log files is tedious and difficult. Furthermore, the size of logs at large companies quickly makes it impossible to store more than a few months of log data. As a result, there has been a several researches done in the field of log processing, so that we can reduce the size of the log file, find important patterns, and present the log messages in a structured format. We divide the research into different categories: clustering-based techniques, AI-based techniques, rule-based techniques, event-based techniques, graph-based techniques, and event abstraction based techniques.

Clustering-Based Log Processing and Abstraction Techniques

Several clustering-based log processing techniques were suggested by researchers [64, 46, 71, 53, 56]. Salfner *et al.* proposed a technique that assigns unique identifiers to log messages on the basis of Levenshtein's edit distance, clusters similar message sequences, and applies a statistical noise filtering algorithm to remove the noise from the clusters [64, 46].

Vaarandi *et al.* [74] proposed a clustering-based log processing algorithm called Simple Log File Clustering Tool (SLCT) that finds frequent patterns in the log file. They build a frequency table that contains the number of times a word occurs in a particular position in the log line. Then to cluster log line and to find important patterns, frequent words from the frequency table are searched in a given log line.

To differentiate between the static and the dynamic parts of the log messages, Nagappan *et al.* [53] suggested a log abstraction technique that leverages the algorithm used in a Simple Log Clustering

Tool (SLCT). They use the algorithm to clusters static and the dynamic part of the log lines. The advantage of this log abstraction technique is that this algorithm can be used in a scenario where we do not have access to the source code. Jiang *et al.* [36] suggested a technique that uses both the source code and log file to perform log abstraction. Their technique involves the following steps: a) anonymization uses heuristics to recognize the dynamic parts in the log lines, b) tokenization puts the anonymized log lines into groups based on the number of words and estimated parameters in each log line, and c) categorization compares the log lines in each group and abstracts them to their corresponding execution events. Jiang *et al.*. use source code during log abstraction and thereby outperform Nagappan *et al.*.

To perform log analysis and a comparison between logs that are generated during the execution of large-scale cloud applications, Shang *et al.* [66] extended the technique suggested by Jiang *et al.* [36]. First, they do log abstraction. Second, they recover the execution sequences of the abstracted log events. Finally, they perform a comparison between the pseudo and large-scale cloud deployments. In all these previous works, the logs under analysis were execution logs [53] and load testing log [36, 66]. In this work, we have adapted the log abstraction approaches to work on test logs.

AI-Based Log Processing Techniques

Many log processing techniques use AI algorithms like neural network, decision tree, and Bayesian network to find patterns in the log file [33, 27, 47, 67, 79]. For example, Wei *et al.* perform log parsing and source code analysis to detect the structure of the log messages. Then they use algorithms like PCA and decision trees to make the log messages more readable for the log operators [82]. Another AI-based algorithm was used by Kenji *et al.* where they use Hidden Markov Model to detect correlated events and anomalous events [84]. Kenji *et al.* use Hidden Markov Model to represent the behavior of dynamic syslog. Finally they calculate an anomaly score for the series of messages using test statistics and if the anomaly score exceeds the threshold then an alarm is raised.

Rule-Based Log Processing Techniques

Log processing can also be performed using rule-based techniques [22, 28, 75]. In rule-based techniques, domain experts use their system knowledge, and construct regular expressions to identify important log messages. The disadvantage of rule-based technique is that it is tedious and requires substantial effort and must be constantly maintained an updated manually.

Event-Based Log Processing Techniques

Another popular log processing technique was proposed by Hellerstein *et al.* [29]. In this technique, they perform pattern discovery with the help of event bursts, periodic events, and mutually dependent events. Where event burst happens when some critical element fails, periodic events happen at regular intervals, and mutually dependent events co-occur.

Graph Theory-Based Log Processing Techniques

Graph theory based log processing technique was proposed by Nagappan *et al.* where they use acyclic directed graphs to detect the pattern of repetitive log lines [54]. Nagappan *et al.* first remove the dynamic content from the log file and identify different events. Finally, they represent each event as nodes in a graph, where the number of transitions between the events represents the weight of the edges.

2.6 Research into categorization of test failures

Large complex systems involve complex test environments that lead to failures that are not product faults. These "false alarms" have received attention because successful classification of false test alarms saves time for testing teams. False alarms can also slow down the development team when test failures stop the build. Herzig *et al.* [32] tackle this issue at Microsoft by automatically detecting false test alarms. They use association rules to classify test failures into false test alarms. The association rules use test failure patterns to perform the classification task. Instead of having two outcomes for a test failure, CAM [35] uses an information retrieval technique to classify test failures into seven categories at Huawei. They train on a large corpus of manually categorized test logs. In our work, we use historical test logs to find specific log lines that tend to be associated with product faults. This allows us to not only ignore false alarms, but to provide likely log line location of the failure.

2.7 Our proposed work

Software testing is a large and a diverse field of study. We focus on using test logs to identify specific log lines that are likely to lead to faults. We know that performing log analysis on huge unstructured logs is difficult. As a result, our initial step is to reduce the size of the log file. We reduce the size of the log file by performing log abstraction, a technique suggested by Shang *et al.* [66]. The size of the failed test log is further reduced by removing all the log lines that also occur in passing test log.

In the next step, we use the historical bug reports, and the occurrence frequencies of the log lines in the past test failures to perform the following tasks: a) flag suspicious log lines in the current failing test log b) predict whether the test failed due to product fault or environmental problem. We introduce three variants for fault identification:SkewCAM, LogLiner, and FaultFlagger. SkewCAM and LogLiner rely on occurrence frequencies of the log lines to carry out their tasks. Whereas, FaultFlagger technique uses both historical bug reports and occurrence frequencies.

Finally, we carry out a comparative study between our CAM, SkewCAM, LogLiner, and FaultFlagger. We evaluate each technique with the help of *FaultsFound* and *LogLinesFlagged*, where *FaultsFound* denotes the percentage of total faults found using a given technique, and *LogLinesFlagged* denotes the percentage of log lines that were flagged as a suspicious statement.

Chapter 3

Ericsson Background and Study Methodology

3.1 Methodology

Discussions with Ericsson developers revealed two challenges in the identification of faults in a failing test log: 1) the complex test environment introduces many non-product test failures and 2) the logs contain an overwhelming amount of detailed information. To overcome these challenges, we perform *log abstraction* to remove contextual information, such as run date and other parameters. Lines that occur in both failing and passing logs are unlikely to reveal a fault, so we perform a *log diff* with the last passing log to remove lines that are not related to the failure. Finally, we extract the rarest log lines and use information retrieval techniques to identify the most likely cause of a fault. We elaborate on each step below.

Listing 3.1: An example of log abstraction. With # representing concrete values that have been abstracted.

```
1. Successfully connects to station
2. Latency at #, above normal
3. No of connection #, higher than normal
4. Perform loadbalancing
5. Latency at #, normal range
6. Testing on # for reliability
```

3.1.1 Log Abstraction

Logs at Ericsson tend to contain a large number of lines, between 1300 and 5800 with a median of 2500 lines. The size makes it difficult for developers to locate the specific line that caused a fault. Log abstraction reduces the number of unique lines in a log. Although the logs do not have a specific format, they contain static and dynamic parts. The dynamic run specific information, such as the date and test machine, can obscure higher level patterns. By removing this information, abstract lines contain the essence of each line without the noisy details.

For example, in Figure 1, the log line “Latency at 50 sec, above normal” contains static and dynamic parts. The static parts describe the high-level task, *i.e.* an above normal latency value. The latency values, are the dynamic parts of the log line, *i.e.* “50" seconds. In another run, we may obtain a “Latency at 51 sec, above normal”. Although both logs contain the same high-level task, without log abstraction these two lines will be treated as different. With log abstraction, the two log lines will record the same warning.

We adapt Shang *et al.*’s [66] log abstraction technique, our approach has the following steps:

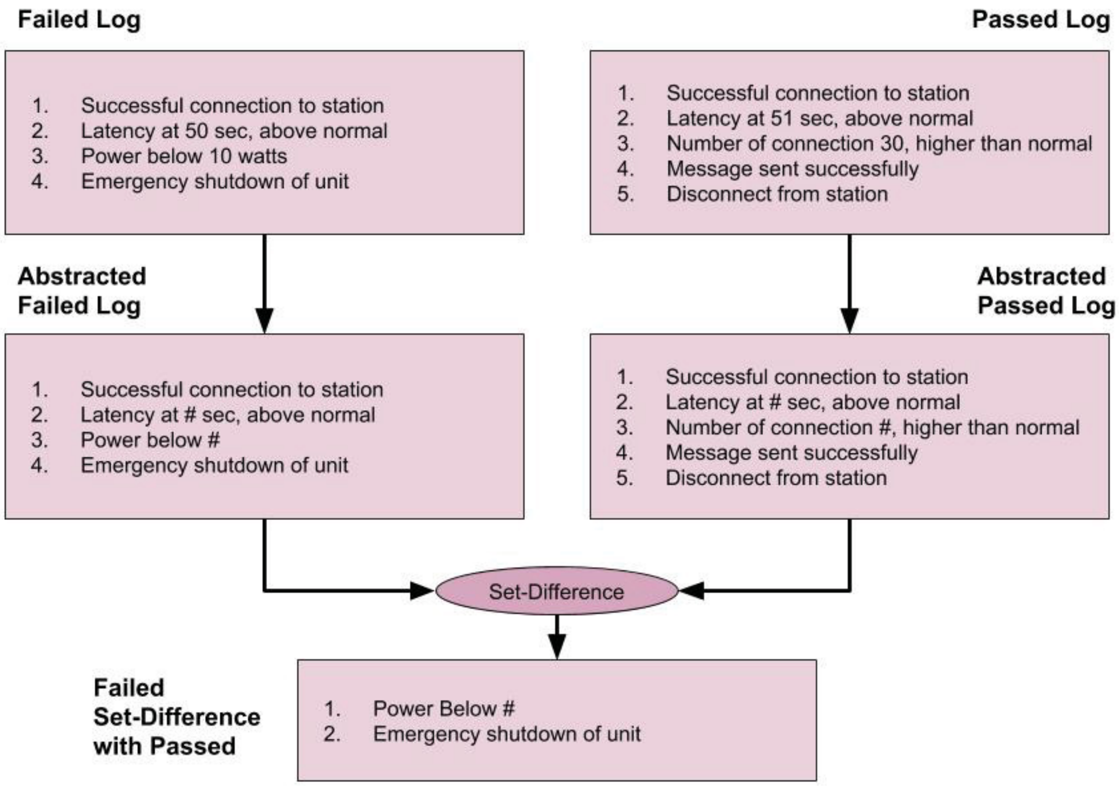
Anonymization: During this step we use heuristics to recognize the dynamic part of the log line. We use heuristics like *StaticVocabulary* to differentiate between the static and the dynamic part of the log line. For example, the test source code contains the log line `print "Latency at %d sec, above normal", latencyValue`. We wrote a parser to find the static parts of the test code, which we store as the *StaticVocabulary*. With the help of StaticVocabulary, we replace the dynamic parts of a log with the # placeholder. In our example, the output of log abstraction would be “Latency at # sec, above normal”.

Unique Event Generation: Finally, we remove the abstracted log lines that occur more than once in the abstract log file. We do this because duplicate log lines represent the same event.

3.1.2 *DiffWithPass*

The location of a fault should be contained in the lines of a failing log. In contrast, a passing log should not contain the lines related to a failure. Lines that occur in both a passing and failing log introduce noise when attempting to find the fault in a failing log. We introduce a novel approach where we remove the lines that occur in the passing log from the failing log. In our example, in Figure 1, the failing log contains an above normal latency. However, the passing log also contains this warning, so it is unlikely that the failure is related to latency. In contrast, the line “Power below 10 watts" occurs only in the failing log, indicating the potential cause for the failure.

Performing a set difference operation with all the previous passing logs is computationally expensive and grows with the number of test runs, $O(n)$. Over a six months period there are 74,621



Failed Log:

1. Successful connection to station
2. Latency at 50 sec, above normal
3. Power below 10 watts
4. Emergency shutdown of unit

Passed Log:

1. Successful connection to station
2. Latency at 51 sec, above normal
3. Number of connection 30, higher than normal
4. Message sent successfully
5. Disconnect from station

Abstracted Failed Log:

1. Successful connection to station
2. Latency at # sec, above normal
3. Power below #
4. Emergency shutdown of unit

Abstracted Passed Log:

1. Successful connection to station
2. Latency at # sec, above normal
3. Number of connection #, higher than normal
4. Message sent successfully
5. Disconnect from station

Failed Set-Difference with Passed:

1. Power Below #
2. Emergency shutdown of unit

Figure 1: *DiffWithPass* stages: First, the logs are abstracted. Second a diff operation is performed between the passing and failing logs. Third, only the lines present in the failing log are kept.

passing and 6,106 failing test runs. For each failure we have to compare with the tests previous passing runs, which would lead to over 455 million comparisons. The number of passes makes comparison impractical. To make our approach scalable, we note that a passing log represents an acceptable state for the system. *We perform a diff of the current failing log with the last passing log.* Computationally, we perform one diff comparison, $O(1)$. This novel approach reduces the number of noisy lines in a log and reduces the storage and computational requirements.

3.1.3 Frequency of test failures and faults

Tests with similar faults should produce similar log lines. For example, when a test fails due to a low power problem it produces the following abstract log line: "Power below # watts." A future failure that produces the same abstract log line will likely have failed due to a low power problem.

Unfortunately, many of log lines are common and occur every time a test fails regardless of the root cause. These noisy log lines do not help in identifying the cause of a specific test failure. In contrast, log lines that are rare and that occur when a bug report is created are likely more useful in fault localization. Our fault location technique operationalizes these ideas by measuring the following:

1. *LineFailCount*: the count of the number of times a log line has been in a failing test.
2. *LineFaultCount*: the count of the number of times a log line has been associated with a reported fault in the bug tracker.

After performing log abstraction and *DiffWithPass*, we store a hash of each failing log line in our database. In Figure 2, we show how we increment the count when a failure occurs and a bug is reported. We see that lines that occur in many failures have low predictive power. For example, “Testcase failed at #” is a common log line that has occurred 76 times out of 80 test failures. In contrast, “Power below #” is a rare log line that occurs 5 times out of 80 failures likely indicating a specific fault when the test falls.

Not all test failures lead to bug reports. As we can see the generic log line “Testcase failed at #" has only been present in 10 failures that ultimately lead to a bug report being filed. In contrast, when the log line “Power below #” occurs, testers have filed a bug report 4 out 5 times. When predicting future potential faults this latter log line clearly has greater predictive power with few false positives.

3.1.4 TF-IDF

In this section, we give a background about TF-IDF, a popular numerical statistics widely used in information retrieval techniques. We discuss about TF-IDF as it is used by CAM to classify test failures.

Identifying faults based on test failures and bug reports is too simplistic. We use Term Frequency by Inverse Document Frequency (TF-IDF) to calculate the importance of a term to a document, *i.e.* log, in a collection [65]. The importance of a term is measured by calculating TF-IDF:

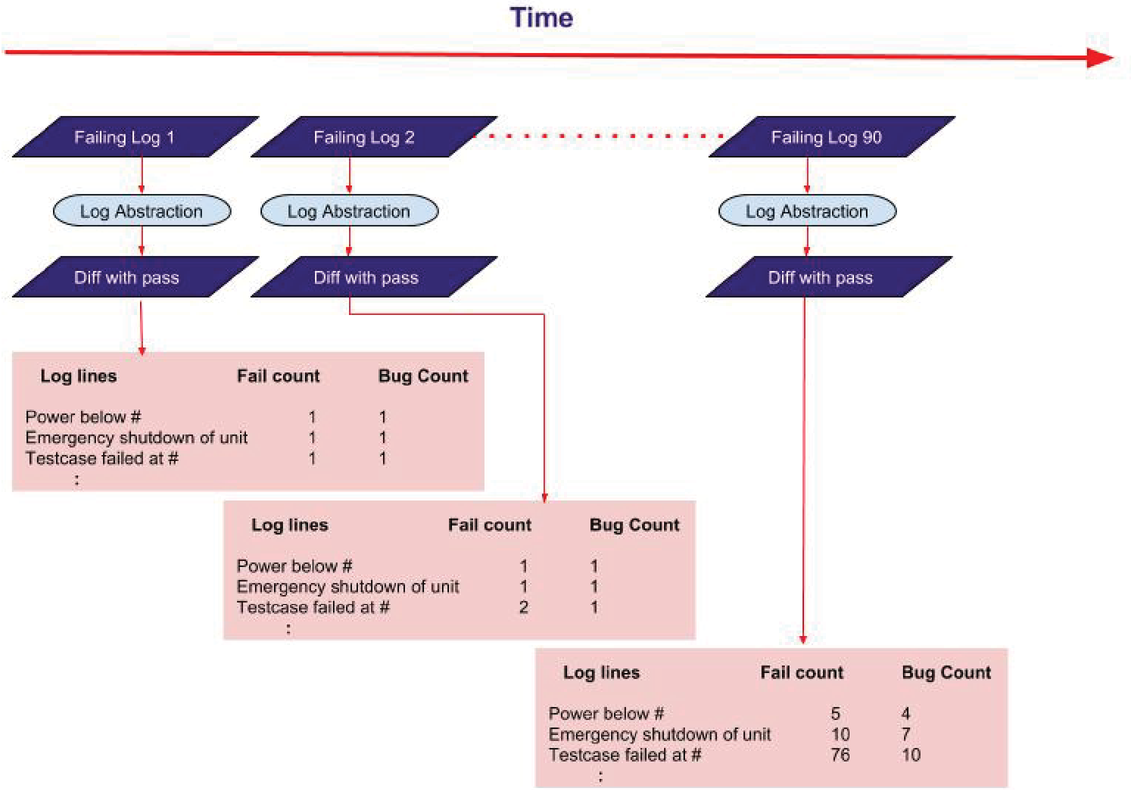
$$TF - IDF_{t,d} = f_{t,d} * \log \frac{N}{N_t} \tag{1}$$

Where $f_{t,d}$ denotes the number of times term t occurred in document d, N denotes the total number of documents in the corpus, and N_t denotes the number of documents that contains the term t [65] [35].

We have discussed in earlier sections that rare log lines are strong indicator of faults. We use IDF (Inverse document frequency) to calculate the importance of a log line to a test log. IDF is defined as:

$$IDF_{l,d} = \log \frac{N}{N_l} \tag{2}$$

Where N denotes the total test logs in our system, and N_l denotes the total number of test logs that contains the log line l.

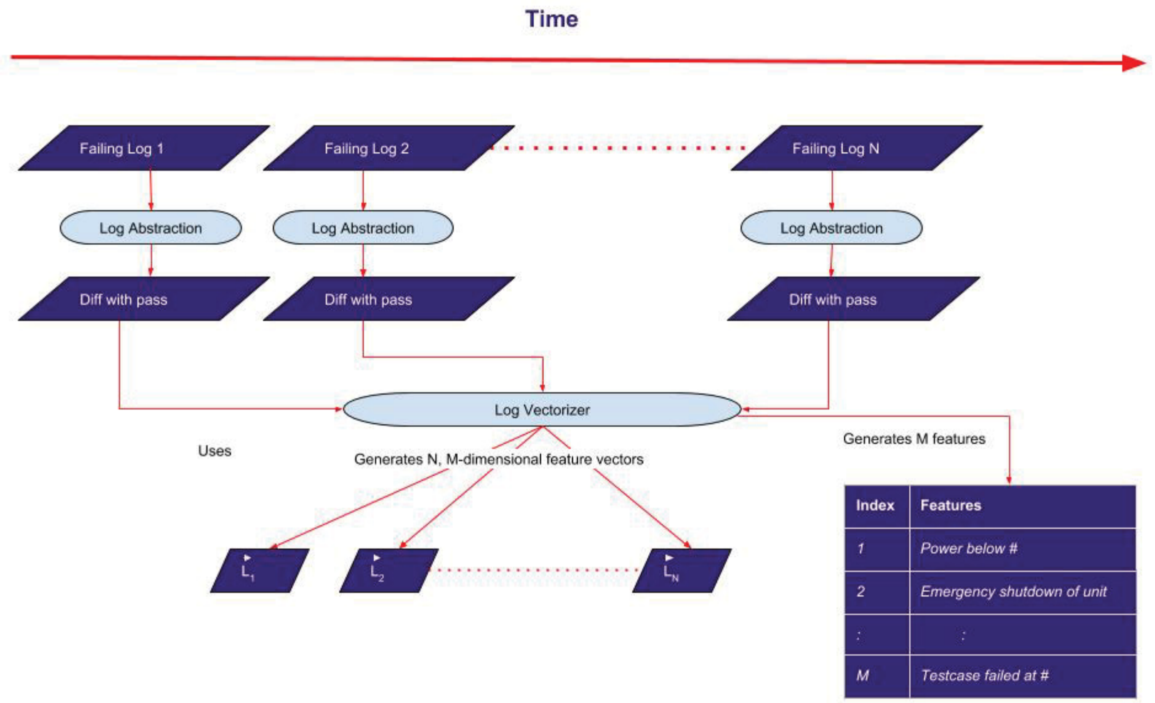


Log lines	Fail count	Bug Count
Power below #	1	1
Emergency shutdown of unit	1	1
Testcase failed at #	1	1
:		

Log lines	Fail count	Bug Count
Power below #	1	1
Emergency shutdown of unit	1	1
Testcase failed at #	2	1
:		

Log lines	Fail count	Bug Count
Power below #	5	4
Emergency shutdown of unit	10	7
Testcase failed at #	76	10
:		

Figure 2: The mapping between the log line failure count and bug report count. Logs lines that have been associated with many bug reports have high predictive power.



Index	Features
1	*Power below #*
2	*Emergency shutdown of unit*
:	:
M	*Testcase failed at #*

Figure 3: Pictorial representation of vector generation from test logs

3.1.5 Log Vectorization

To find similar log patterns that have occurred in the past we transform each log into a vector. Each failed log is represented as a vector and the log lines in our vocabulary denotes the features of these vectors. For example, if we have N failed logs in our system then we would generate N vectors, a vector for every failed log. The dimension of the vectors is determined by the number of unique log lines in our corpus. If we have M unique log lines then the generated vectors would be M-dimensional. Figure 3 shows the pictorial representation of the entire process.

Many techniques exist to assign values to the features[1] of the vectors. In our experiment, we use two techniques. The first technique assigns value to the feature of a vector $\vec{L}_1$ with the help of log line IDF (Inverse document frequency) as shown in the following formula:

$$Feature_{i,FailedLog_1} = \log \frac{N}{N_i} \tag{3}$$

Where, $\vec{L}_1$ denotes the feature vector of $FailedLog_1$, $Feature_{i,FailedLog_1}$ denotes the i^{th} feature of $FailedLog_1$, N denotes the total number of failed logs, and N_i denotes the number of failed logs that contains the $Feature_i$ (log line).

The second technique assigns a value to the feature of a vector $\vec{L}_1$ with the help of log line IDF

[1] Feature represents a unique log line

and *LineFaultCount* as shown in the following formula:

$$Feature_{i,FailedLog_1} = (LineFaultCount_i + 1) * \log \frac{N}{N_i} \tag{4}$$

Where $LineFaultCount_i$ denotes the number of bug reports associated with a feature $Feature_i$ (log line).

3.1.6 Cosine Similarity

To find similar logs and log lines to predict faults we use cosine similarity. It is defined as[65] [62]:

$$similarity = \cos\theta = \frac{\vec{L_1} \cdot \vec{L_2}}{\|L_1\|_2 \|L_2\|_2} \tag{5}$$

Where L_1 and L_2 represent the feature vectors of two different test logs. We represent each past failing log and current failing log as vectors, and compute the cosine similarity between the vector of current failing log and the vectors of all the past failing logs.

During the calculation of cosine similarity we only take top N log lines (features) from the vector of current failing log. Since our prediction is based only on these lines we consider these N lines to be flagged for further investigations. We are able to predict not only which log will lead to product faults, but also which lines are most likely the cause of those faults.

3.1.7 Exclusive K Nearest Neighbours (***EKNN***)

To determine whether the current log will lead to a bug report, we modify the K nearest neighbours (KNN) approach as follows. For the distance function, we use the cosine similarity of the top N lines as described above. For the voting function, we need to consider the skew in our dataset. Our distribution is highly skewed because of the significant proportion of environmental failures. We adopt an extreme scheme whereby if any of the K nearest neighbours has lead to a bug report in the past, we predict that the current test failure will lead to a bug report. If none of the K neighbours has lead to a past bug report, then we predict no fault. This approach is consistent with our overriding goal of finding as many faults as possible, but may lead to additional log lines being flagged for inspection.

To set the value of K, we examine the distribution of test failures and measure the performance of different values of K from 1 to 120.

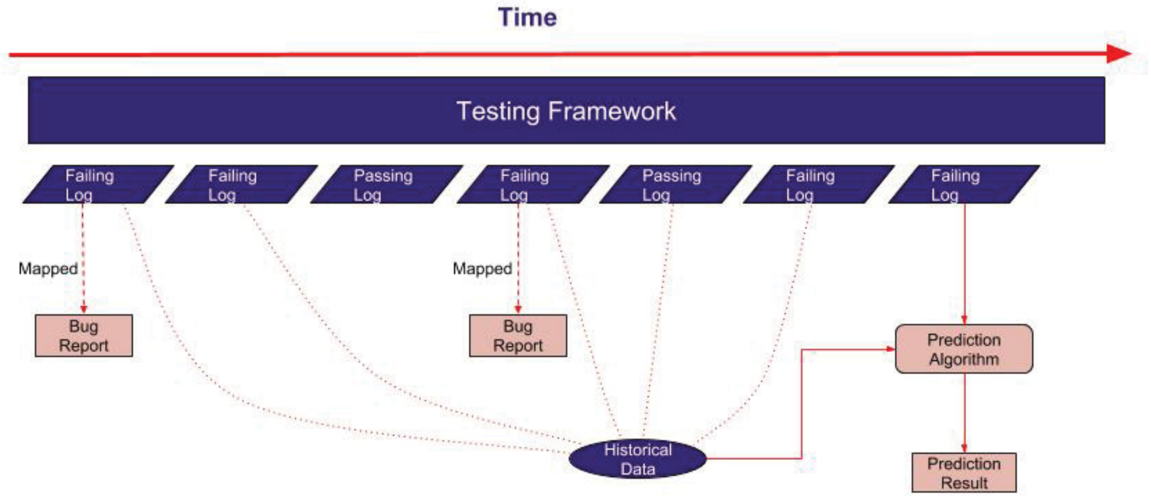


Figure 4: Historical logs are used to predict future log failures.

3.2 Evaluation Setup

Ericsson testers evaluate test failures on a daily basis. We run a simulation to show how accurate our daily prediction would have been. Figure 4 shows the incremental framework that we use to train and test each fault location technique [32, 35, 83, 7]. Our simulation period runs for 6 months from February 2017 to July 2017. We train and test the approaches on the nightly software test runs for day D =0 to D =T. To predict whether a failure on day D =t will reveal a product fault, we train on the historical data from D =0 to D =t-1 and test on D =t. Similarly, to predict on day D =t + 1, we train on the historical data from D =0 to D =t. We repeat this training and testing cycle for each nightly run until we reach D =T.

Figure 5 shows how our technique trains and predicts product faults. The fault prediction technique uses historical test data to predict whether the current test failure is due to a product fault or an environmental fault.

Our goal is to capture the maximum number of product bugs reported while inspecting the minimum number of log lines. We operationalize this goal by calculating the percentage of *FaultsFound* and the percentage of *LogLinesFlagged*. We define *FaultsFound* and *LogLinesFlagged* as the following:

$$FaultsFound = \frac{TotalPredictedFault}{TotalActualReportedFault} * 100 \tag{6}$$

$$LogLinesFlagged = \frac{TotalLogLinesFlagged}{TotalLogLinesInAlllogs} * 100 \tag{7}$$

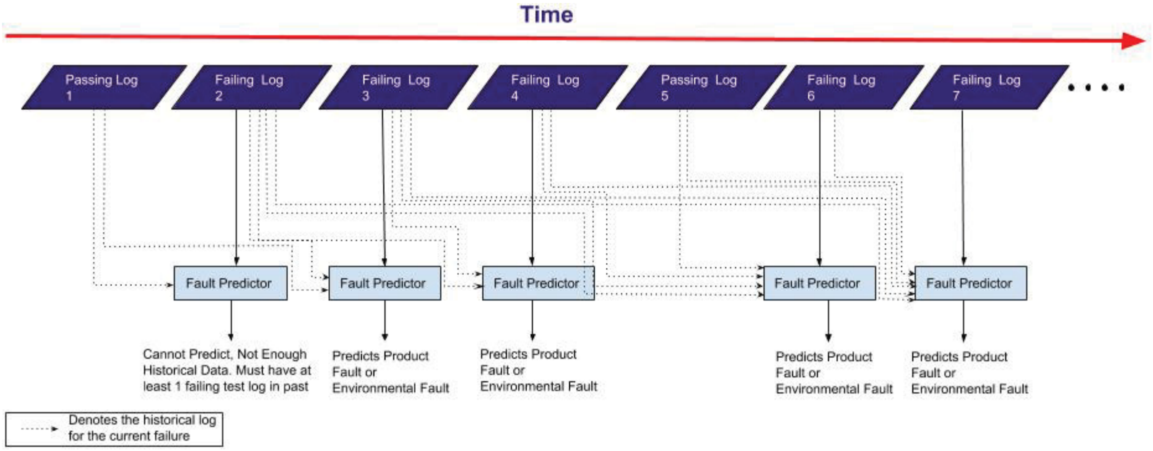

Time
Passing Log 1
Failing Log 2
Failing Log 3
Failing Log 4
Passing Log 5
Failing Log 6
Failing Log 7
Fault Predictor
Fault Predictor
Fault Predictor
Fault Predictor
Fault Predictor
Cannot Predict, Not Enough Historical Data. Must have at least 1 failing test log in past
Predicts Product Fault or Environmental Fault
Predicts Product Fault or Environmental Fault
Predicts Product Fault or Environmental Fault
Predicts Product Fault or Environmental Fault
Denotes the historical log for the current failure

Figure 5: A pictorial representation of fault prediction

3.3 Ericsson Test Process and Data

At Ericsson there are multiple levels of testing from low level developer run unit tests to expensive simulations of real world scenarios on hardware. In this thesis, we focus on integration tests at Ericsson. Testers are responsible for running and investigating integration test failures. Our goal is to help these testers quickly locate the fault in a failing test log.

Integration testing is divided into test suites that contain individual tests. In Figure 6, we illustrate the integration testing at Ericsson. There are multiple levels of integration testing. The passing builds are sent to the next level of integration tests. For each integration test case, *TestID*, we record the *TestExecutionID* which links to the result *LogID* and the verdict. The log contains the runtime information that is output by the build that is under test. For each failing test, we store the log and also store the previous passing run of the test for future comparison with the failing log. Failing tests that are tracked to a product fault are recorded in the bug tracker with a *TroubleReportID*. Environmental and flaky tests do not get recorded in the bug tracker and involve re-testing once the environment has been fixed. We study a six month period from Feb 2017 to July 2017.

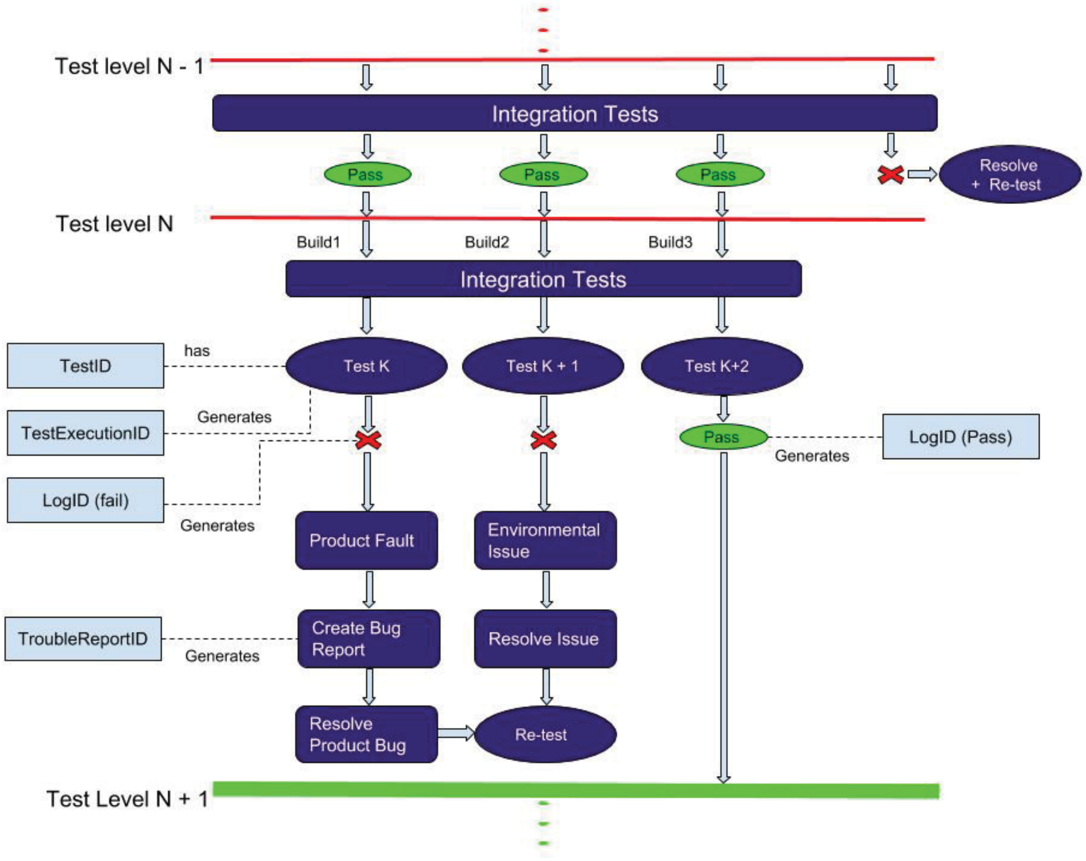


Figure 6: The Ericsson integration test process. We study the integration testing stage. Testing has already gone through earlier developer testing stages (N-1) and will continue to later integration stages (N+1). The ID, *e.g.,* TestID, show the data entities that we extract.

Chapter 4

Results and Tool: Fault Prediction and Localization

Discussions with Ericsson developers revealed two challenges in the identification of faults in a failing test log: 1) the complex test environment introduces many non-product test failures and 2) the logs contain an overwhelming amount of detailed information. To solve these problems, we mine the test logs to predict which test failures will lead to product faults and which lines in those logs are most likely to reveal the cause of the fault. In the previous chapter, we discussed the background on the four techniques: CAM, SkewCAM, LogLiner, and FaultFlagger. We also described our evaluation metrics: the number of faults found, *FaultsFound*, and the number of log lines investigated to find those faults, *LogLinesFlagged*. In this chapter, we present our results.

Table 1: CAM: TF-IDF & KNN

K	% FaultCaught	% LogLineFlagged	% CorrectlyFlaggedProductFaults	Execution Time (mins)
1	47.30	4.13	67.04	420
15	**50**	**4.38**	**66.10**	**444**
30	47.23	4.36	67.60	458
60	47.14	4.07	67.47	481
120	47.43	4.23	68.10	494

Table 2: SkewCAM: CAM with *EKNN*

K	% FaultCaught	% LogLineFlagged	% CorrectlyFlaggedProductFaults	Execution Time (mins)
1	47.13	4.21	67.04	190
15	86.65	21.18	27.09	199
30	**88.64**	**27.71**	**18.77**	**204**
60	90.84	38.10	16.40	223
120	90.84	43.65	14.66	253

Table 3: LogLiner: Line-IDF & *EKNN*

K	N	% FaultCaught	% LogLineFlagged	% CorrectlyFlaggedProductFaults	Execution Time (mins)
1	1	47.23	0.06	75.23	30
15	1	67.48	0.14	42.55	46
30	1	68.22	0.16	38.14	52
60	1	68.22	0.17	36.01	68
120	1	68.22	0.17	35.67	91
1	10	47.27	0.56	77.47	39
15	10	82.35	2.39	33.54	85
30	**10**	**84.60**	**2.98**	**25.85**	**90**
60	10	86.05	3.92	21.15	98
120	10	86.05	4.26	19.50	127

Table 4: FaultFlagger: PastFaults $*$ Line-iDF & EKNN

K	N	% FaultCaught	% LogLineFlagged	% CorrectlyFlaggedProductFaults	Execution Time (mins)
1	1	53.10	0.06	81.50	36
15	1	87.33	0.33	23.82	49
30	**1**	**88.88**	**0.42**	**18.52**	**54**
60	1	90.41	0.54	15.35	83
120	1	90.41	0.58	14.38	119
1	10	63.00	0.80	72.79	48
15	10	88.45	3.23	25.12	88
30	10	89.20	3.99	19.28	103
60	10	90.84	5.39	15.41	124
120	10	90.84	6.04	13.89	185

4.1 Result 1. CAM: TF-IDF & KNN

CAM has successfully been used at Huawei to categorize test logs [35]. We re-implement their technique and perform a replication on Ericsson test logs. We discussed the data processing steps in Section 3.1. We then apply TF-IDF to the terms in each failing log. Cosine similarity is used to compare the current failing log with *all* past failing logs for a test. CAM then calculates a threshold to determine if the current failing log is similar to any of the past logs. The details can be found in their paper and we use the same threshold value of similarity at $t = .7$. If the value is below the threshold, then KNN is used for classification. CAM sets $K = 15$ [35], we vary the number of neighbours from $K = 1$ to 120.

Table 1 shows that the direct application of CAM to the Ericsson dataset only finds 50% or fewer of the product faults. We also see that increasing the value of K neighbours does not increase the number of *FaultsFound*. For example, at $K = 15$ the CAM finds 50% of the product faults. However, when we increase K to 30 it only captures 48% of the product faults. The table also shows the percentage of total flagged logs that were correctly predicted as product faults, *CorrectlyFlaggedProductFaults*. We noticed that increasing the value of K does not lead to a significant increase in the percentage of *CorrectlyFlaggedProductFaults*. The percentage varies between 66% and 68% for $K = 1$ to 120.

CAM is also computationally expensive and on average it takes 7 hours to process the entire dataset. There are two main factors that contribute to this computational cost. First, the CAM performs word based TF-IDF which generates large feature vectors and then calculates the cosine similarity between the vector of current failing log and the vectors of all the past failing logs. The time complexity is $O(|V| \cdot |L|)$. Second, the algorithm computes a similarity threshold using the past failing logs that increases computational time by $O(|V| \cdot |l|)$. Where V denotes the vocabulary of terms present in the failing test logs, L denotes the total number of failing test logs, and l denotes a smaller set of failing test logs used during the calculation of similarity threshold.

CAM finds 50% of the total faults. CAM flags the entire failing log for investigation.

4.2 Result 2. SkewCAM: CAM with *EKNN*

Ericsson's test environment involves complex hardware simulations of cellular base stations. As a result, many test failures are environmental and do not lead to a product fault. Since the data is skewed, we modify KNN. In Section 3.1.7, we define Exclusive KNN (*EKNN*) to predict a fault if any of the K nearest neighbours has been associated with a fault in the past.

We adjust CAM for skewed data. Like CAM, SkewCAM uses TF-IDF to vectorize each log and cosine similarity to compare the current failing log with all previously failing logs. However, we remove the threshold calculation as both the study on CAM [35] and our experiments show that it has little impact on the quality of clusters. Instead of using KNN for clustering SkewCAM uses *EKNN*. We vary the number of neighbours from $K = 1$ to 120.

Table 2 shows that more neighbours catch more product faults but also flags many lines. At $k = 30$, SkewCAM catches 89% of the all product faults, but flags 28% of the total log lines. Interestingly as we increase K to 120 the number of faults found increased to only 91%, but the lines flagged increases to 44%. Table 2 also shows the percentage of total flagged logs that were correctly predicted as product faults, *CorrectlyFlaggedProductFaults*. We noticed that as we increase the value of K to 120, the percentage of *CorrectlyFlaggedProductFaults* reduces drastically. At $K = 30$, only 19% of the total flagged logs were correctly predicted as product faults.

Adjusting CAM for skewed data by using *EKNN* allows SkewCAM to catch most product faults. However, the improvement in the number of *FaultsFound* comes at the cost of flagging many more lines for inspection. Testers must now face the prospect of investigating many log lines.

Despite removing the threshold calculation, SkewCAM is still computationally expensive because like CAM it applies word based TF-IDF. Hence, it has a time complexity of $O(|V| \cdot |L|)$.

SkewCAM finds 89% of the total faults, but flags 28% total log lines for inspection. It is also computationally expensive.

4.3 Result 3. LogLiner: Line-IDF & *EKNN*

SkewCAM can accurately identify the logs that lead to product faults, however it flags a large number of suspicious log lines that need to be examined by testers.

To effectively identify product faults while flagging as few log lines as possible, we developed novel technique called LogLiner. LogLiner uses the uniqueness of log lines to predict product faults. We calculate the uniqueness of the log line by calculating the Inverse Document Frequency (IDF) for each log line. Before calculating IDF, we remove run-specific information from logs by performing data processing as explained in Section 3.1.

IDF is used to generate the vectors for the current failing log and all of the past failing logs according to the equation below. For each unique line in a log we calculate its IDF score according to the following:

$$IDF(Line) = \log \frac{TotalNumLogs}{FrequencyOfLogLine} \tag{3}$$

In order to reduce the number of flagged log lines, we perform our prediction using the top IDF scoring N lines from the current failing log. We then apply cosine similarity and compare with the K neighbours using *EKNN* to predict whether the current failing test log will lead to fault.

During our experiment, we varied K from 1 to 120, N from 1 to 10, and studied the relationship between the number of neighbours (K), top N lines with highest IDF score, percentage *FaultsFound*, and percentage *LogLinesFlagged*.

Table 3 shows the impact of changing these parameters. Low parameter values $N = 1$ and $K = 1$ lead to *FaultsFound* at 47% with only < 1% of *LogLinesFlagged*. By using the top line in a log and examining the result for the top neighbour, we are able to perform at similar levels to CAM. CAM and SkewCAM use all the log lines during prediction. With this setting LogLiner finds 88% of the faults, but flags 29% of the lines, a similar result to SkewCAM.

Setting LogLiner to more reasonable values, $K = 30$ and $N = 10$, we are able to find 85% of the faults by flagging 3% of the log lines for inspection. Drastically increasing $K = 120$ and keeping $N = 10$ we find 86% of the faults but flag 4% of the lines.

The table also shows the percentage of total flagged logs that were correctly predicted as product faults, *CorrectlyFlaggedProductFaults*. For, $N = 1$ and $N = 10$, we noticed that increasing the value of K leads to a significant reduction in the percentage of *CorrectlyFlaggedProductFaults*. At $K = 30$ and $N = 10$, LogLiner is only to correctly predict 26% of the total flagged logs.

> LogLiner finds 85% of the total faults while flagging only 3% of the total log lines for inspection.

4.4 Result 4. FaultFlagger: PastFaults $*$ Line-IDF & EKNN

LogLiner flags fewer lines, but drops slightly in the number of *FaultsFound*. We build on LogLiner with FaultFlagger which incorporates faults into the line level prediction.

IDF is usually weighted. Instead of using a generic weight, such as term frequency, we use the number of times a log line has been associated with a product fault in the past. We add 1 to this frequency to ensure that the standard IDF of the line is applied if a line has never been associated with any faults. We weight line-IDF with the Fault Frequency (FF) according to the following equation:

$$\begin{aligned} \text{FF-IDF}(\mathit{Line}) &= (\mathit{LineFaultCount} + 1) * IDF(\mathit{Line}) \\ &= (\mathit{LineFaultCount} + 1) * log\frac{\mathit{TotalNumLogs}}{\mathit{FrequencyOfLogLine}} \end{aligned} \tag{8}$$

As with the previous approaches, we vary the number of neighbours from $K = 1$ to 120 and the number of top lines flagged with $N = 1$ and 10. Table 4 shows that the value of N has little impact on the number of faults found. Furthermore, the number of *FaultsFound* increases only slightly after $K \geq 15$. As a result, we use $N = 1$ and $K = 30$ for further comparisons and find that FaultFlagger find 89% of the total faults with 0.5% of total log lines flagged for inspection.

In Table 4, we can see the percentage of total flagged logs that were correctly predicted as product faults, *CorrectlyFlaggedProductFaults* by FaultFlagger. The percentage of *CorrectlyFlaggedProductFaults* shows a similar pattern as SkewCAM and LogLiner as the percentage of *CorrectlyFlaggedProductFaults* reduces drastically with the increase in the value of K. However, FaultFlagger performs comparatively better than CAM, SkewCAM and LogLiner at $K = 1$ and $N = 1$, and was able to correctly predict 82% of the total flagged logs.

Compared to SkewCAM, FaultFlagger finds the same number of faults, but SkewCAM flags 28% of total log lines compared FaultFlagger $< 1\%$. Compared to LogLiner, FaultFlagger finds 4 percentage points more faults with 2.5 percentage points fewer lines flagged.

> FaultFlagger finds 89% of the total faults and flags only 0.5% of lines for inspection.

4.5 Implementing the FaultFlagger Tool at Ericsson

We implemented our best fault identification algorithm, FaultFlagger, as a tool for Ericsson testers. The tool uses historical test logs and bug reports to perform predictions on the test failures on a daily basis. In Section 4.5.1, we briefly describe the architecture of the tool. The features of the tool has been demonstrated in Section 4.5.2.

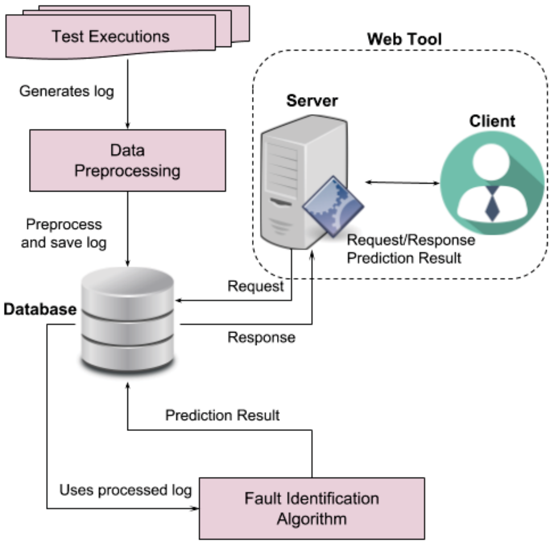


Figure 7: Architecture diagram

4.5.1 Architecture

During software testing various test cases are executed periodically to validate the quality of the software product. During these executions test cases produce test execution logs. We use these test execution logs and predict whether a test failed due to a *product fault* or an *environmental fault*.

Figure 7 shows the architecture diagram of our tool. With the help of the architecture diagram, we can see that our foremost step is the preprocessing of the execution log. During the preprocessing step we clean the log and compute the following things:

1. *LineFailCount*: the count of the number of times a log line has been in a failing test.

2. *LineFaultCount*: the count of the number of times a log line has been associated with a product fault.

After processing the log, we store the processed log, *LineFailCount*, *LineFaultCount* and other meta-data in the database. Later we apply our fault identification algorithm that uses Inverse-Document-Frequency (IDF), *LineFaultCount*, Cosine Similarity, and KNN on the processed logs and predicts whether a test failed due to a *product fault* or an *environmental fault*. The aforementioned steps take place periodically with the help of a python script and a Linux cron job.

In the architecture diagram, we can also see a web tool which is a client server application. With the help of the web tool a client (software developers and testers) can find out whether a given failure is due to a *product fault* or an *environmental fault*. The web tool also flags the potential log lines that carry crucial information about the test failure. In the following section we delineate the features of our web tool.

4.5.2 Tool Features

Daily Fault Prediction Summary

At Ericsson, testing teams execute a bundle of test cases every night on the nightly build. On the subsequent day, testers manually examine all the failed test cases and their corresponding execution logs. The manual inspection of the test log helps in identifying the actual reason behind the test failure, which could either be a *product fault* or an *environmental fault*. If the reason behind the test failure is a *product fault* then a TR (Trouble Report/Bug Report) is generated by a tester. The entire process relies heavily on the manual inspection of the test log which takes a considerable amount of time, and consequently increases the cost of software testing.

To save testers time and reduce software testing cost, we created a tool that automatically predicts whether a test has failed due to a *product fault* or an *environmental fault*. The tool displays a daily fault prediction summary on its home page, which can be easily accessed by a tester via a web browser.

The daily fault prediction summary contains a list of all the failed test cases that were executed during the testing of a nightly build. In Figure 8, we can see the home page of our tool, where *DAILY TROUBLE REPORT FORECAST* section of the home page shows a daily fault prediction summary. The prediction summary contains the following columns:

1. *STP* is an identifier of the equipment where the test was executed.

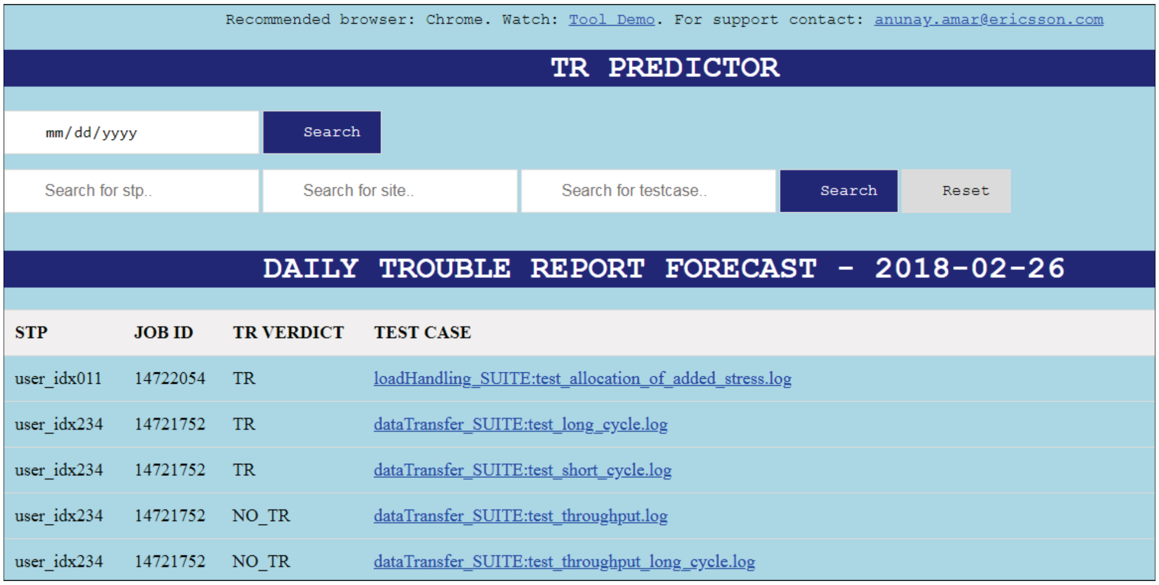
2. *JOB ID* is an unique identifier that represents the test execution id.

3. *TR VERDICT* shows the prediction result. If a test case is anticipated to have failed due to a *product fault* then it displays *TR* as a verdict, otherwise it shows *NO_TR*

4. *TEST CASE* displays the name of the failed test case.

We use the *TR Verdict* column to sort the rows present in the fault prediction summary. Rows that have *TR*s come before the rows that do not have TRs (*NO_TR*). This improves the usability of the tool by allowing a tester to quickly identify all the test cases that could lead to an actual TR (Trouble Report).

The daily fault prediction summary also helps software testers in prioritizing their daily work. The testers can use the summary to start their inspection from those test cases for which the tool has predicted a *TR*.



Recommended browser: Chrome. Watch: Tool Demo. For support contact: anunay.amar@ericsson.com

TR PREDICTOR

DAILY TROUBLE REPORT FORECAST - 2018-02-26

STP	JOB ID	TR VERDICT	TEST CASE
user_idx011	14722054	TR	loadHandling_SUITE:test_allocation_of_added_stress.log
user_idx234	14721752	TR	dataTransfer_SUITE:test_long_cycle.log
user_idx234	14721752	TR	dataTransfer_SUITE:test_short_cycle.log
user_idx234	14721752	NO_TR	dataTransfer_SUITE:test_throughput.log
user_idx234	14721752	NO_TR	dataTransfer_SUITE:test_throughput_long_cycle.log

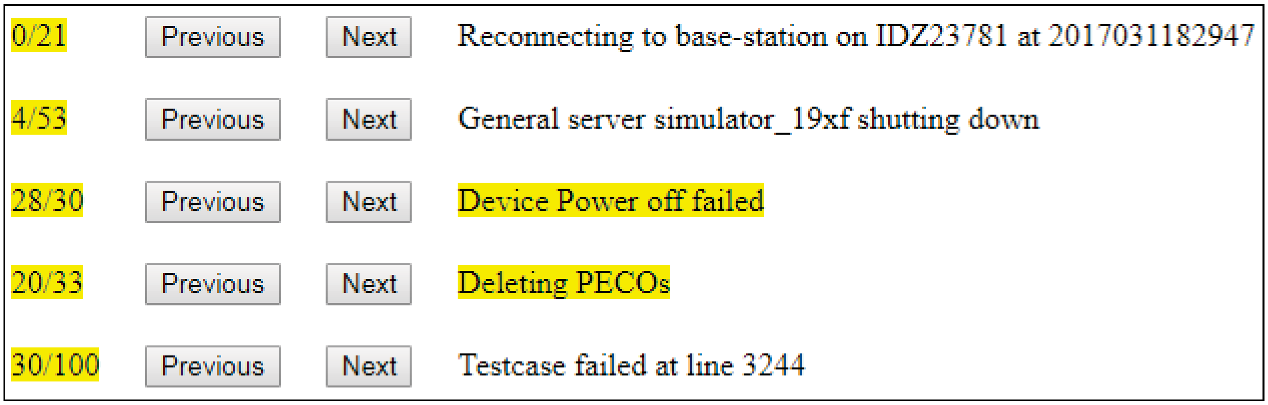
Figure 8: Tool home page

Test Log With a TR

Test logs are overwhelming large. To solve this problem we customized the test execution log and added information that would help the developers/testers in quickly identifying the cause of the test failure. We added *LineFailCount* and *LineFaultCount* values in our customized test execution log. These values are represented as a ratio of *LineFaultCount* and *LineFailCount*. Each ratio tells us how many times does a log line has lead to a TR(fault) in the past and how often does it occur in a failing test log. Using this ratio a developer can identify important log lines that carry crucial information about the test failure. For example, using *LineFaultCount*, developers can identify log lines that frequently occur when a test fails due to a *product fault*. Furthermore, developers can use *LineFailCount* to identify those log lines that occur rarely or has occurred due to an unseen product fault.

Our tool highlights the N most important log lines in the test execution log that carry crucial information about the test failure. Important log lines are identified with the help of our fault identification algorithm. The algorithm uses IDF score and *LineFaultCount* to identify the important log lines.

Figure 9 shows the log lines generated during the test failure. The highlighted lines shown in the figure represent the top N lines that our algorithm has identified as most likely to indicate the cause of the fault. The ratio in the left margin shows the LineFaultCount over the LineFailCount. For example, we can see that the generic line “Reconnecting to base-station on IDZ23781 at 2017031182947” has a ratio of 0/21. This means that when this line is present in a failed log there were zero trouble reports



```
0/21     Previous   Next   Reconnecting to base-station on IDZ23781 at 2017031182947
4/53     Previous   Next   General server simulator_19xf shutting down
28/30    Previous   Next   Device Power off failed
20/33    Previous   Next   Deleting PECOs
30/100   Previous   Next   Testcase failed at line 3244
```

Figure 9: Customized execution log when the prediction is a TR

filed in 21 test failures. Clearly this line has little predictive power. Interestingly, although this line is abstracted when making the prediction, the run specific information such as "201731182947" is displayed in the tool to further help developers in identifying the specific, for example, node that was running the test.

In contrast, we can see that there is a line "Device Power off failed" which has a ratio of 28/30. This ratio implies that the line was present 30 times in the test failures, and out of 30 times, 28 times there was trouble report filed against the test failure. This ratio shows a strong relationship between trouble report and test failure, and as a result the corresponding log line is considered as a potential reason behind the test failure. Similarly, "Deleting PECOs" also has a high LineFaultCount over LineFailCount ratio and is flagged as a potential reason behind the test failure.

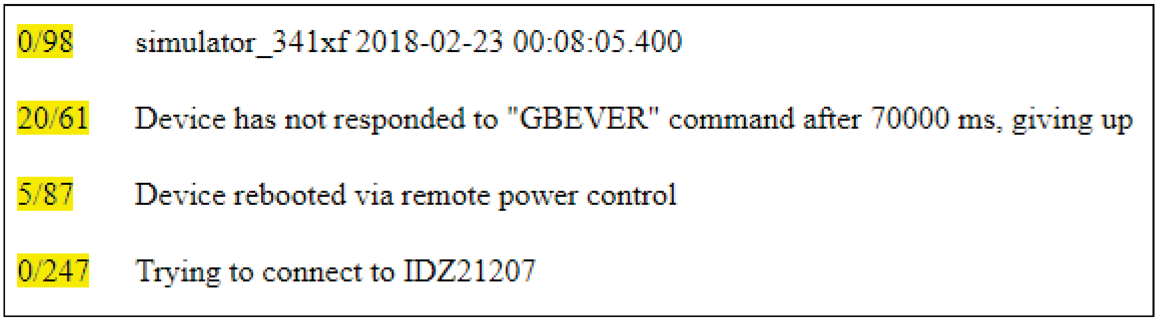
For ease in the navigation between highlighted log lines we have also provided a next and previous buttons.

Test Log Without a TR

In this section we would learn how we represent a failed test log in the scenario when our algorithm predicts that the failure is due to an *environmental fault* i.e. when *TR_VERDICT* for the corresponding failed test case is *NO_TR*.

When our algorithm predicts that the test has failed due to an *environmental fault*, we show a customized test execution log to the developers for manual inspection. The customization is performed by adding ratios of *LineFaultCount* and *LineFailCount* to the execution log. As explained in the previous section, we add these ratios because *LineFaultCount* and the *LineFailCount* help developers in identifying important log lines that carry crucial information about the failure.

In Figure 10, we can see a customized test execution log which is generated when a test fails and our tool predicts that the failure is due to an *environmental fault*.



```
0/98      simulator_341xf 2018-02-23 00:08:05.400

20/61     Device has not responded to "GBEVER" command after 70000 ms, giving up

5/87      Device rebooted via remote power control

0/247     Trying to connect to IDZ21207
```

Figure 10: Customized execution log when the prediction is not a TR

Miscellaneous Filters

To improve the usability of our tool we provided filters. Figure 8 shows the filters: a) Date Filter b) STP Filter c) Site Filter d) Testcase Filter.

If a developer wants to look at the test cases that failed on a specific date then the developer can select a date from the Date filter to get the result. To find out all the failed test cases that were executed on a specific equipment, developer can use the STP filter. Similarly, to find out all the test cases that were executed at specific location, we can use the Site filter. Finally, if a developer wants to look at all the test executions of a particular test case then he can use the Testcase filter.

Chapter 5

Discussion and Threats

5.1 Discussion

Our discussion revolves around the number of correctly identified test failures that lead to faults, *FaultsFound* in Figure11, and the number log lines used to make the prediction, *i.e.* the lines that are flagged for manual investigation, *LogLinesFlagged*, in Figure 12. Testers want to catch a maximal number of faults while investigating as few flagging lines as possible.

CAM technique: We re-implemented CAM [35] and evaluated it on a new dataset. CAM is based on a popular information retrieval technique. The technique uses simple word based TF-IDF to represent failed test log as vectors. Then it ranks the past failures with the help of their corresponding cosine similarity score. Finally, it uses KNN to determine whether the current test failure is due to a product fault and presents its finding to the testers. CAM has two major limitations. First, CAM does not flag individual log lines that are the likely cause of the fault. Instead it only categories test failures into, for example, product vs environmental failure. As a result, CAM forces testers to manually examine the entire log file to find the important log lines that carry crucial information about the test failure. The second limitation is that CAM performs poorly on the Ericsson dataset, see Figure 11 and 12. We can see that even when we increase the number of K neighbours, the number of *FaultsFound* does not increase and stays around 50%. CAM performs poorly because the Ericsson data is highly skewed due to the significant proportion of environmental failures, which reduces the effectiveness of voting in KNN.

SkewCAM technique: We modify CAM for skewed datasets. SkewCAM uses an exclusive, *EKNN*, strategy that we designed for skewed data. If any of the nearest K neighbours has had a fault in the past, SkewCAM will flag the log as a product fault. Figure 11 shows that SkewCAM finds 89% of the product faults solving the first limitation of CAM. SkewCAM's major limitation

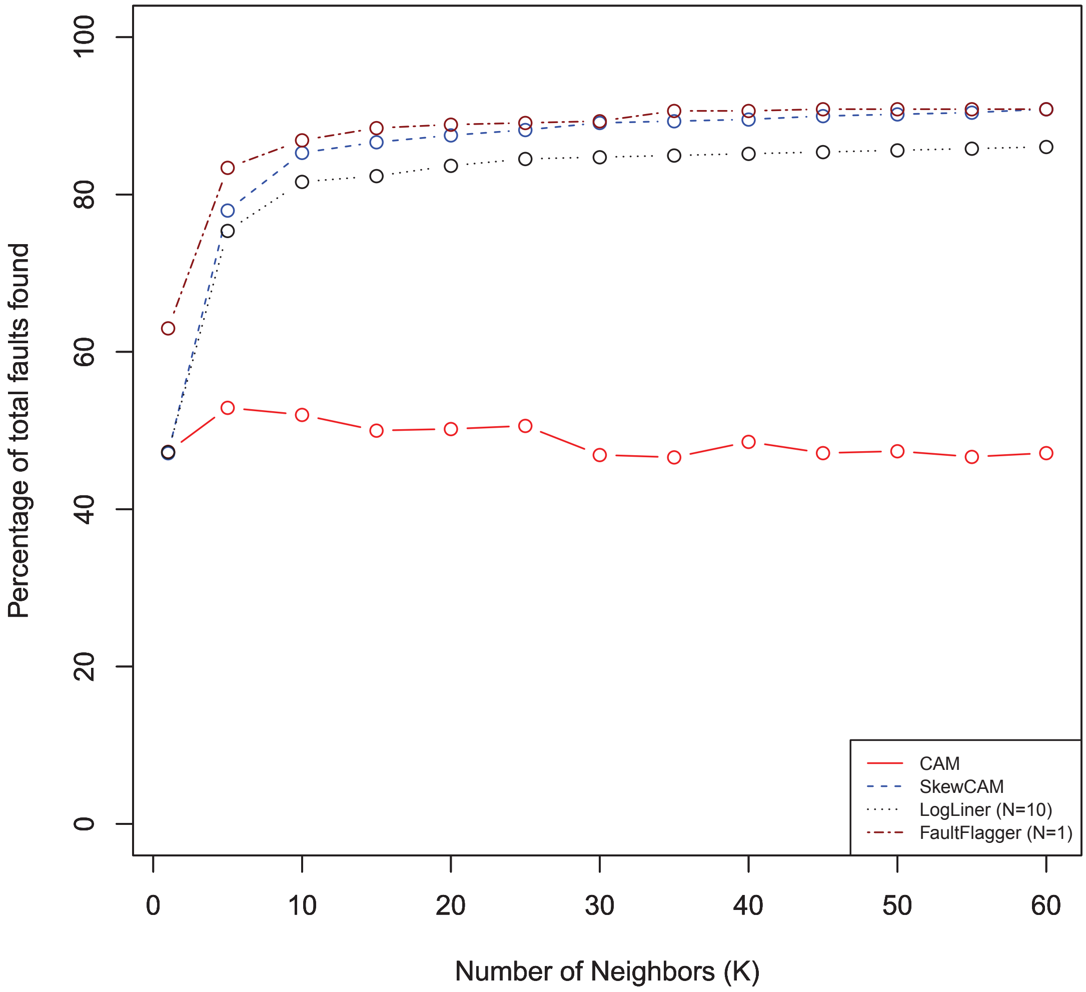


Figure 11: *FaultsFound* with varying K. SkewCAM and FaultFlagger find a similar number of faults.

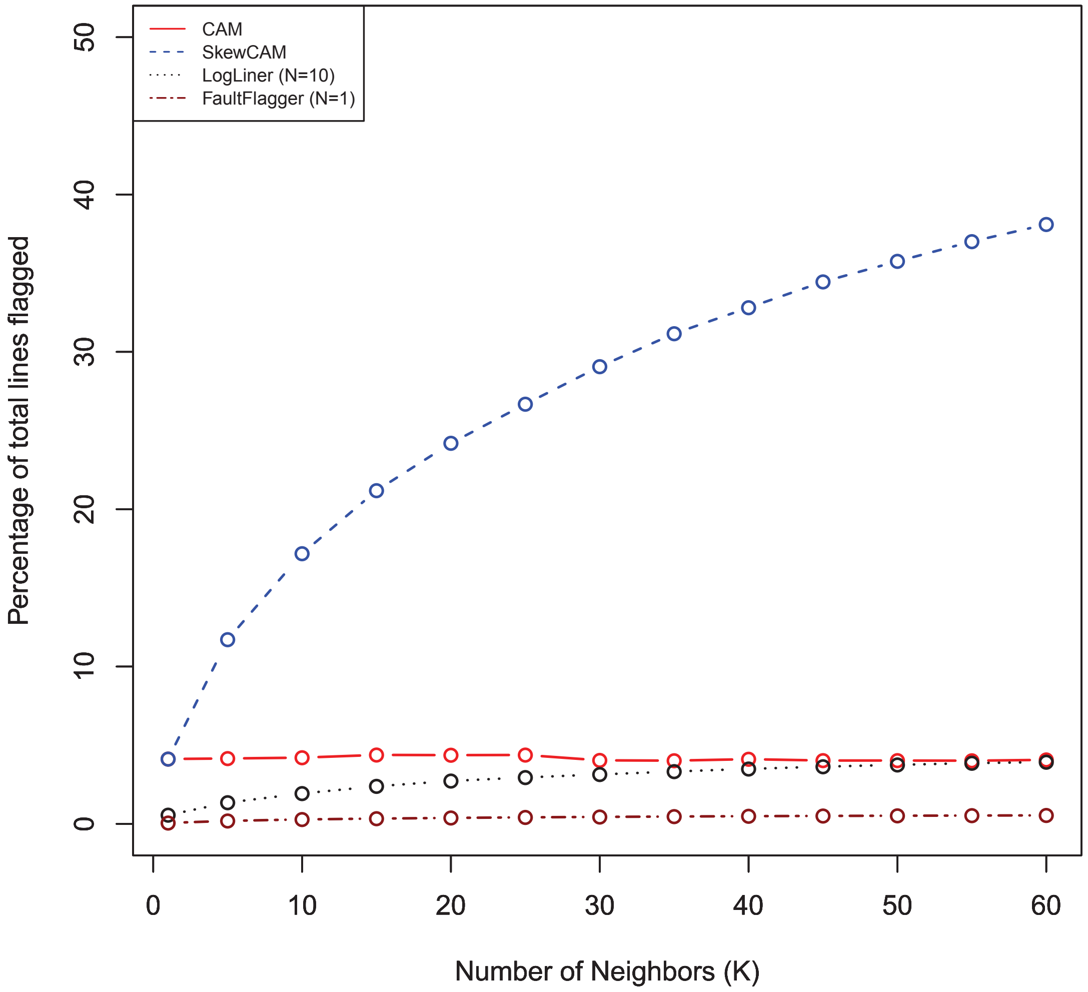


Figure 12: *LogLinesFlagged* with varying k. SkewCAM flags an increasing number of lines, while FaultFlagger remains constant around 1% of total log lines.

is that it flags 28% of all log lines in making its fault predictions. As a result, testers must manually examine many log files to identify the cause of the failure. Figure 12 also shows that as the number of K neighbours increases so too does the number of *LogLinesFlagged*. Like CAM, SkewCAM uses the entire failed log in its prediction providing poor fault localization within a log.

LogLiner technique: To reduce the number of *LogLinesFlagged*, we introduce a novel technique called LogLiner. LogLiner makes a novel modification to TF-IDF by employing IDF at the line level. The IDF score helps to identify rare log lines in the current failing log. Our conjecture is that rare lines are indicative of anomalies, which in turn, indicate faults. LogLiner selects top N most rare log lines in the current failing log. These N lines are vectorized and used to calculate the similarity with past failing test logs. *Since only N lines are used in the prediction, only N lines are flagged for investigation by developers drastically reducing the manual effort in fault localization.*

LogLiner identifies 85% of the faults by flagging only 3% of the lines. In Figure 12 we see that LogLiner flags many fewer lines than SkewCAM, 27% *LogLinesFlagged*, and that the number of lines flagged remains nearly constant with changes in K. However, Figure 11 shows that LogLiner finds slightly fewer product faults than SkewCAM, 89% *FaultsFound*. While we have reduced the number of lines flagged from 27% to 3%, LogLiner has reduced the fault finding effectiveness from 89% to 85%.

FaultFlagger technique: To improve the number of *FaultsFound* and reduce the number of *LogLinesFlagged*, we suggest a novel technique called FaultFlagger that uses the association between log lines and *LineFaultCount*. FaultFlagger uses LogLiner's line based IDF score and *LineFaultCount* to represent log files as vectors. We then select the top N log lines that are both rare and associated with the most historical faults. Our experimental result shows that the log line rarity and its association with fault count is a strong predictor of future product faults. Figures 11 and 12 show that FaultFlagger finds 89% of the faults while flagging only 0.5% of the total log lines for investigation. This is an improvement compared to LogLiner's 3% *LogLinesFlagged*. In comparison to SkewCAM, FaultFlagger finds the same number of faults, but SkewCAM flags 27% of the lines compared to the less than 1% flagged by FaultFlagger.

5.2 Performance and Log Storage

In this section we compare the performance and storage overhead associated with CAM, SkewCAM, LogLiner, and FaultFlagger. Table 1, Table 2, Table 3, and Table 4 shows the execution time of CAM, SkewCAM, LogLiner, and FaultFlagger respectively. We can see that both CAM and SkewCAM are computationally more expensive than LogLiner and FaultFlagger. At $K = 30$, CAM, SkewCAM, LogLiner (N=10) and FaultFlagger (N=10) take 458 minutes, 204 minutes, 90 minutes, and 54 minutes respectively to analysis six months worth of log files. CAM and SkewCAM are slower as they both perform word based TF-IDF which generates large feature vectors as a result they have a time complexity of $O(|V| \cdot |L|)$, where V denotes the vocabulary of log lines present in the failing test logs, and L denotes the total number of failing test logs. In contrast, LogLiner and FaultFlagger line based IDF which uses a small subset of the total log lines as a result LogLiner and FaultFlagger have a time complexity of $O(|v| \cdot |L|)$, where v denotes a small subset of the vocabulary V.

Performing log analysis on huge log files is tedious and expensive. CAM, SkewCAM, LogLiner, and FaultFlagger all requires historical test logs for fault prediction and localization. As a result, we are required to store the test logs for a long period of time which increases the storage overhead. To ameliorate the storage overhead, we reduce the size of the raw log files by performing log abstraction and set-difference. Over a one month period, we calculate the amount of reduction in the overall log storage size. We found that with log abstraction we can reduce the log storage size by 78%. When we employ both log abstraction and set-difference we were able to reduce the log storage size by 94%. This reduction drastically reduces the storage requirements and allows companies to store longer test logs for a longer time period.

5.3 Generalizing to Other Organizations

Our techniques, FaultFlagger and LogLiner can be generalized and applied to test data generated by other organizations. FaultFlagger uses historical test data to perform fault prediction and localization. As a result, we need the following information to apply our fault prediction and localization technique to different test datasets. First, we need a history of trouble reports/bug reports. The report should be correctly linked to test failures caused by product faults. Second, FaultFlagger uses historical test logs to perform the prediction and analysis. As a result, we also need the organizations to store the test execution log for several months. Storing test logs can be costly for organizations as the logs are huge in size. However, we can overcome this problem by applying log abstraction technique that drastically reduces the size of the log file. Third, as our log processing technique performs a set difference operation between a current failing log and the most recent passing log, we also require the organizations to store the most recent passing log.

5.4 Threats to Validity

In our case study at Ericsson, the test failure data is highly skewed because of the significant proportion of environmental failures. As a result, we use large number of neighbours, $K = 30$. If the technique were applied to other projects, the historical distribution of product faults to test failures would need to be calculated. It is then simple to adjust the value of K based on the number of faults that lead to bug reports. Further study is necessary to determine the efficacy of the technique on other projects with different failure distributions.

Our fault identification techniques use log abstraction to pre-process the log files. During the log abstraction process, we lose run-time specific information from the test log. Though the run-time specific information can help in the process of fault identification it adds substantial noise and increased log size. We reduce the size of the log and increase the fault localization by performing log abstraction. However, we leave the run specific information in when the tester views the log in the FaultFlagger tool so that they can find, for example, which specific node the test has failed upon.

Although we can find 89% of all faults, we cannot predict all the product faults because the reason for all failures is not contained in the log, *i.e.* not all run information is logged. Furthermore, when a test fails for the first time we cannot calculate a line IDF score or calculate the cosine similarity with previously failing neighbours. We found that predicting first time test failures as a product fault leads to many false positives at Ericsson. As a result, in this work, a first test failure has no neighbours and so we predict that there will be no product fault. This parameter can easily be adjusted for other projects.

Chapter 6

Conclusion

6.1 Related Work

In this section, we position our work in the context of the literature on log abstraction and categorizing test failures.

6.1.1 Log Processing and Abstraction

The size of logs at large companies quickly makes it impossible to store more than a few months of log data. Log abstraction finds the dynamic and the static part of the log and converts the log lines into unique events. Conversion of logs into unique events leads to a significant reduction in the overall size of the log file. Nagappan *et al.* [56] suggested a log abstraction technique that leverages the algorithm used by a popular Simple Log Clustering Tool (SLCT). They use the algorithm to clusters static and the dynamic part of the log lines. The advantage of this log abstraction technique is that this algorithm can be used in a scenario where we do not have access to the source code. Jiang *et al.* [36] suggested a technique that uses both the source code and log file to perform log abstraction. Their technique involves the following steps: a) anonymization uses heuristics to recognize the dynamic parts in the log lines, b) tokenization puts the anonymized log lines into groups based on the number of words and estimated parameters in each log line, and c) categorization compares the log lines in each group and abstracts them to their corresponding execution events. Jiang *et al.* use source code during log abstraction and thereby outperform Nagappan *et al.*.

To perform log analysis and a comparison between logs that are generated during the execution of large-scale cloud applications, Shang *et al.* [66] extended the technique suggested by Jiang *et al.* [35]. First, they do log abstraction. Second, they recover the execution sequences of the abstracted log events. Finally, they perform a comparison between the pseudo and large-scale cloud deployments.

We adapt Jiang *et al.*'s technique to abstract Ericsson test log. In all these previous works, the logs under analysis were execution logs [56] and load testing log [36, 66]. In this work, we have adapted the log abstraction approaches to work on test logs.

6.1.2 Categorizing Test Failures

Large complex systems involve complex test environments that lead to failures that are not product faults. These "false alarms" have received attention because successful classification of false test alarms saves time for testing teams. False alarms can also slow down the development team when test failures stop the build. Herzig *et al.* [32] tackle this issue at Microsoft by automatically detecting false test alarms. They use association rules to classify test failures into false test alarms. The association rules use test failure patterns to perform the classification task. In contrast, we use historical test logs to find specific log lines that tend to be associated with product faults. This allows us to not only ignore false alarms, but to provide likely log line location of the failure.

Instead of having two outcomes for a test failure, CAM [35] uses an information retrieval technique to classify test failures into seven categories at Huawei. They train on a large corpus of manually categorized test logs. Our work could be extended to other categories provided that Ericsson developer classified logs at a finer granularity than product and environmental fault.

6.2 Concluding Remarks

We have developed a tool and technique called FaultFlagger that can identify 89% of the faults while flagging less than 1% of the total log lines for investigation by developers. While developing FaultFlagger we make three major novel contributions.

First, previous works attempted to reduce the size of the log files and find potential fault patterns using log abstraction techniques [36, 56, 66]. In contrast, we diff the the current failing log with the last passing log. Our observation is that the location of a fault should be contained in the lines of a failing log, while the last passing log should not contain the lines related to a failure. Lines that occur in both a passing and failing log introduce noise when attempting to find the fault in a failing log. As a result, we remove the lines that occur in the passing log from the failing log. There are three advantages to our *DiffWithPass* technique. (1) we only need to store the failing log and the most recent passing log, (2) diffing further reduces the size of the failed test log which makes storing logs easier and cheaper, and (3) the technique reduces the noise present in the failed test log because all the lines in the last pass are removed.

Second, our initial discussion with testers revealed that they want to find the most faults while fewest log lines possible. We evaluate each technique on the basis of *FaultsFound* and *LogLinesFlagged*.

Previous works can only classify test failures based on logs and do not flag specific log lines as potential causes [36]. Testers must manually go through the entire log file to identify the log lines that are causing the test failure. In order to predict product fault and locate suspicious log lines, we introduce a novel approach where we train our model on a subset of log lines that occur in current failing test log. FaultFlagger identifies the most specific lines that have lead to past faults, *i.e.* PastFaults * Line-IDF + *EKNN*. In our Ericsson tool, FaultFlagger highlights the flagged lines in the log for further investigation.

Third, FaultFlagger drastically outperforms the state of the art, CAM [35]. CAM finds 50% of the total faults. CAM flags the entire failing log for investigation. When CAM is adjusted for skewed data, SkewCAM, it is able to find 89% of the total faults, as many FaultFlagger, however, it flags 28% of the log lines compared to the less than 1% flagged by FaultFlagger.

Chapter 7

Appendix

In this appendix we show the results for larger variations in K and N as well as precision and recall instead of *FaultsFound* and *LogLinesFlagged*.

Total log files flagged:

Table 5 shows the percentage of total log files flagged by CAM, SkewCAM, LogLiner, and FaultFlagger. The result shows that the number of neighbours K has a little effect on CAM. For $K = 1$ to 120, CAM flags 4.2% of the total log files as product faults. Whereas, the number of log files flagged as product faults by SkewCAM, LogLiner, and FaultFlagger increase drastically with the increase in the value of K. For $K = 1$ to 120, LogLiner and FaultFlagger flag comparatively smaller number of log files than SkewCAM. In contrast, when we compare LogLiner with FaultFlagger, we noticed that for $K = 1$ to 120, LogLiner always flags fewer log files as product faults.

Correctly flagged log files:

In Table 6 we can see the percentage of log files that were correctly flagged as product faults by CAM, SkewCAM, LogLiner, and FaultFlagger. For $K = 1$, FaultFlagger (N=1) flags the highest percentage of correctly flagged log files as product faults. We noticed that increasing the value of K leads to a significant reduction in the percentage of correctly flagged log files as product fault by SkewCAM, FaultFlagger (N=1), and FaultFlagger (N=10). In contrast, CAM does not vary much with K, and staying around 67%.

Median precision and recall:

Table 7 and Table 8 show the median precision and recall of fault prediction techniques, namely, CAM, SkewCAM, LogLiner, FaultFlagger. We use median precision and median recall instead of precision and recall because prediction performance of the technique can vary with test cases. For some specific test cases, precision and recall are 100% and for other they are 0%. As a result, we use median precision and median recall.

Unique faults found:

Some product faults appear more than once and as a result they are always linked with the same trouble report. In table 9, we show the percentage of unique faults found by CAM, SkewCAM, LogLiner, FaultFlagger (N=1), and FaultFlagger (N=10). We noticed that for SkewCAM, LogLiner and FaultFlagger the percentage of unique faults found increase with the increase in the value of K. In contrast, the percentage of unique faults found by CAM does not vary much with the value of K, and stays in the range of 59.01% and 62%. We also noticed that FaultFlagger at $K = 30$, finds the maximum number of unique product faults and is 91.66%.

Table 5: Percentage of Total Log Files Flagged by CAM, SkewCAM, LogLiner (N=10), FaultFlagger (N=1) and FaultFlagger (N=10)

K	CAM	SkewCAM	LogLiner (N=10)	FaultFlagger (N=1)	FaultFlagger (N=10)
1	4.26	4.26	2.26	2.42	3.23
5	4.27	8.53	5.46	7.71	7.69
10	4.27	11.59	7.81	11.44	10.92
15	4.41	13.84	9.67	13.61	13.07
20	4.42	16.93	11.03	15.40	14.68
25	4.40	18.27	11.92	16.82	16.04
27	4.41	19.44	13.65	16.98	16.13
30	4.22	20.37	12.70	17.90	17.23
35	4.21	21.29	13.45	18.92	18.25
40	4.26	22.03	14.12	19.84	19.24
45	4.21	22.73	14.70	20.48	20.05
50	4.21	22.71	15.16	20.97	20.64
55	4.20	22.73	15.61	21.44	21.26
60	4.23	22.36	15.87	21.87	21.79
120	4.22	26.46	17.22	23.44	24.42

Table 6: Percentage of Correctly Flagged Log Files by CAM, SkewCAM, LogLiner (N=10), FaultFlagger (N=1) and FaultFlagger (N=10)

K	CAM	SkewCAM	LogLiner (N=10)	FaultFlagger (N=1)	FaultFlagger (N=10)
1	67.04	67.04	77.47	81.50	72.79
5	66.17	42.14	53.94	38.20	40.26
10	66.18	32.23	40.57	27.70	29.35
15	66.10	27.09	33.54	23.82	25.12
20	66.81	22.34	29.78	21.33	22.37
25	67.37	20.86	27.55	19.53	20.71
27	67.54	19.63	25.21	19.53	20.65
30	67.60	18.77	25.85	18.52	19.28
35	68.10	18.02	24.69	17.55	18.39
40	67.98	17.46	23.51	16.73	17.45
45	68.16	17.37	22.83	16.39	16.74
50	67.72	16.91	22.14	16.01	16.27
55	67.10	16.95	21.51	15.66	15.79
60	67.47	16.40	21.15	15.35	15.41
120	68.10	14.66	19.50	14.38	13.89

Table 7: Median Precision Percentage of CAM, SkewCAM, LogLiner (N=10), FaultFlagger (N=1) and FaultFlagger (N=10)

K	CAM	SkewCAM	LogLiner (N=10)	FaultFlagger (N=1)	FaultFlagger (N=10)
1	50.00	50.00	55.55	50.00	50.00
5	50.00	28.99	33.33	17.64	21.98
10	50.00	18.06	28.57	15.89	15.76
15	50.00	13.63	20.00	13.30	13.63
20	50.00	13.69	20.00	12.12	12.23
25	55.55	12.13	15.38	11.73	11.76
27	55.55	12.04	15.38	11.76	11.76
30	55.05	11.76	15.38	10.90	11.76
35	68.10	10.63	15.38	10.52	11.47
40	55.01	10.10	15.38	9.83	10.86
45	55.01	9.83	14.70	9.83	10.10
50	58.33	9.83	14.28	9.30	10.00
55	60.41	9.83	14.28	8.92	9.83
60	52.72	9.42	14.28	8.92	9.83
120	60.41	8.92	14.28	8.92	8.92

Table 8: Median Recall Percentage of CAM, SkewCAM, LogLiner (N=10), FaultFlagger (N=1) and FaultFlagger (N=10)

K	CAM	SkewCAM	LogLiner (N=10)	FaultFlagger (N=1)	FaultFlagger (N=10)
1	60.00	60.00	60.00	66.66	66.66
5	55.55	77.55	75.00	75.00	75.00
10	55.55	86.60	77.77	85.71	86.60
15	55.55	87.50	77.77	87.50	87.50
20	55.55	87.41	77.77	87.50	88.88
25	59.09	87.50	77.77	87.50	88.88
27	59.09	87.90	80.00	87.50	88.88
30	59.45	88.88	80.00	88.88	88.88
35	55.05	88.88	80.00	88.88	88.88
40	63.06	88.88	80.00	88.88	88.88
45	57.32	88.88	80.00	88.88	88.88
50	60.00	88.88	80.00	88.88	88.88
55	59.23	88.88	80.00	88.88	88.88
60	55.05	88.88	80.00	88.88	88.88
120	57.46	88.88	80.00	88.88	88.88

Table 9: Unique Faults Found Percentage of CAM, SkewCAM, LogLiner (N=10), FaultFlagger (N=1) and FaultFlagger (N=10)

K	CAM	SkewCAM	LogLiner (N=10)	FaultFlagger (N=1)	FaultFlagger (N=10)
1	62.00	62.00	52.77	58.33	63.88
5	63.32	76.88	72.22	80.55	86.11
10	63.33	84.28	75.00	84.21	88.88
15	63.68	86.01	75.00	86.40	88.88
20	61.23	86.01	77.77	86.40	88.88
25	61.23	87.34	77.77	86.40	88.88
27	60.00	87.34	77.77	88.88	88.88
30	60.00	88.88	77.77	88.88	91.66
35	60.00	88.88	77.77	88.88	91.66
40	59.01	88.88	77.77	88.88	91.66
45	59.01	88.88	77.77	88.88	91.66
50	59.01	88.88	77.77	88.88	91.66
55	59.01	88.88	77.77	88.88	91.66
60	59.01	88.88	77.77	88.88	91.66
120	59.01	88.88	77.77	88.88	91.66

Bibliography

[1] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[2] E. Arisholm, L. C. Briand, and E. B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1):2–17, 2010.

[3] C. T. Bailey and W. L. Dingee. A software study using halstead metrics. In *Proceedings of the 1981 ACM Workshop/Symposium on Measurement and Evaluation of Software Quality*, pages 189–197, New York, NY, USA, 1981. ACM.

[4] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1319–1360, 1994.

[5] V. R. Basili and R. W. Selby. Comparing the effectiveness of software testing strategies. *IEEE transactions on software engineering*, (12):1278–1296, 1987.

[6] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, FOSE '07, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[7] P. Bhattacharya and I. Neamtiu. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[8] S. Biffl, A. Aurum, B. Boehm, H. Erdogmus, and P. Grünbacher. *Value-based software engineering*. Springer Science & Business Media, 2006.

[9] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.

[10] L. C. Briand, Y. Labiche, and Y. Wang. An investigation of graph-based class integration test order strategies. *IEEE Transactions on Software Engineering*, 29(7):594–607, July 2003.

[11] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *Proceedings. 26th International Conference on Software Engineering*, pages 480–490, May 2004.

[12] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, pages 34–44. IEEE Computer Society, 2009.

[13] H. Cleve and A. Zeller. Locating causes of program failures. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 342–351. IEEE, 2005.

[14] D. S. Coutant, S. Meloy, and M. Ruscetta. Doc: A practical approach to source-level debugging of globally optimized code. *ACM SIGPLAN Notices*, 23(7):125–134, 1988.

[15] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 31–41. IEEE, 2010.

[16] V. Debroy, W. E. Wong, X. Xu, and B. Choi. A grouping-based strategy to improve the effectiveness of fault localization techniques. In *Quality Software (QSIC), 2010 10th International Conference on*, pages 13–22. IEEE, 2010.

[17] R. A. DeMillo, H. Pan, and E. H. Spafford. Critical slicing for software fault localization. In *ACM SIGSOFT Software Engineering Notes*, volume 21, pages 121–134. ACM, 1996.

[18] H. Do and G. Rothermel. A controlled experiment assessing test case prioritization techniques via mutation faults. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 411–420. IEEE, 2005.

[19] J. C. Edwards. Method, system, and program for logging statements to monitor execution of a program, Mar. 25 2003. US Patent 6,539,501.

[20] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 235–245. ACM, 2014.

[21] K. O. Elish and M. O. Elish. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, 81(5):649–660, 2008.

[22] P. Fröhlich, W. Nejdl, M. Schroeder, C. Damásio, and L. M. Pereira. Using extended logic programming for alarm-correlation in cellular phone networks. In *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, pages 343–352. Springer, 1999.

[23] B. Ghotra, S. McIntosh, and A. E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 789–800. IEEE Press, 2015.

[24] S. S. Ghuman. Software testing techniques. 2014.

[25] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[26] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[27] D. W. Gurer, I. Khan, R. Ogier, and R. Keffer. An artificial intelligence approach to network fault management. *Sri international*, 86, 1996.

[28] S. E. Hansen and E. T. Atkins. Automated system monitoring and notification with swatch. In *LISA*, volume 93, pages 145–152, 1993.

[29] J. L. Hellerstein, S. Ma, and C.-S. Perng. Discovering actionable patterns in event data. *IBM Systems Journal*, 41(3):475–493, 2002.

[30] K. Herzig. Using pre-release test failures to build early post-release defect prediction models. In *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*, pages 300–311. IEEE, 2014.

[31] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy. The art of testing less without sacrificing quality. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 483–493. IEEE Press, 2015.

[32] K. Herzig and N. Nagappan. Empirically detecting false test alarms using association rules. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 39–48, Piscataway, NJ, USA, 2015. IEEE Press.

[33] J.-F. Huard and A. A. Lazar. Fault isolation based on decision-theoretic troubleshooting. 1996.

[34] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International*

Conference on Software Engineering, ICSE '94, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[35] H. Jiang, X. Li, Z. Yang, and J. Xuan. What causes my test alarm?: Automatic cause analysis for test alarms in system and integration testing. In *Proceedings of the 39th International Conference on Software Engineering*, pages 712–723. IEEE Press, 2017.

[36] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. An automated approach for abstracting execution logs to execution events. *Journal of Software: Evolution and Process*, 20(4):249–267, 2008.

[37] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282. ACM, 2005.

[38] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, June 2013.

[39] Y. Kastro and A. B. Bener. A defect prediction method for software versioning. *Software Quality Journal*, 16(4):543–562, 2008.

[40] T. M. Khoshgoftaar, K. Gao, and N. Seliya. Attribute selection and imbalanced data: Problems in software defect prediction. In *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*, volume 1, pages 137–144. IEEE, 2010.

[41] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 119–129. IEEE, 2002.

[42] S. Kim, E. J. W. Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, March 2008.

[43] P. Knab, M. Pinzger, and A. Bernstein. Predicting defect densities in source code files with decision tree learners. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 119–125. ACM, 2006.

[44] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE transactions on software engineering*, 30(6):418–421, 2004.

[45] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering*, 7(1):49–76, 2002.

[46] C. Lim, N. Singh, and S. Yajnik. A log mining approach to failure analysis of enterprise telephony systems. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 398–403. IEEE, 2008.

[47] A. Lin. *A hybrid approach to fault diagnosis in network and system management*. Hewlett Packard Laboratories, 1998.

[48] L. Luo. Software testing techniques. *Institute for software research international Carnegie mellon university Pittsburgh, PA*, 15232(1-19):19, 2001.

[49] T. Mende and R. Koschke. Effort-aware defect prediction models. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 107–116. IEEE, 2010.

[50] T. Mende, R. Koschke, and M. Leszak. Evaluating defect prediction models for a large evolving software system. In *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pages 247–250. IEEE, 2009.

[51] J. Micco. Flaky tests at google and how we mitigate them. https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html, May 2016.

[52] J. Moeyersoms, E. J. de Fortuny, K. Dejaeger, B. Baesens, and D. Martens. Comprehensible software fault and effort prediction: A data mining approach. *Journal of Systems and Software*, 100:80–90, 2015.

[53] M. Nagappan. Analysis of execution log files. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 2, pages 409–412. IEEE, 2010.

[54] M. Nagappan and B. Robinson. Creating operational profiles of software systems by transforming their log files to directed cyclic graphs. In *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*, pages 54–57. ACM, 2011.

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

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

[57] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 529–540. ACM, 2007.

[58] S. P. Ng, T. Murnane, K. Reed, D. Grant, and T. Y. Chen. A preliminary survey on software testing practices in australia. In *2004 Australian Software Engineering Conference. Proceedings.*, pages 116–125, 2004.

[59] O. Niese. An integrated approach to testing complex systems. 2003.

[60] A. Okutan and O. T. Yıldız. Software defect prediction using bayesian networks. *Empirical Software Engineering*, 19(1):154–181, Feb 2014.

[61] J. Pan and L. T. Center. Procedures for reducing the size of coverage-based test sets. In *Proceedings of International Conference on Testing Computer Software*. Citeseer, 1995.

[62] F. Pop, J. Kołodziej, and B. Di Martino. *Resource Management for Big Data Platforms: Algorithms, Modelling, and High-Performance Computing Techniques*. Springer, 2016.

[63] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on software engineering*, 21(1):19–31, 1995.

[64] F. Salfner and S. Tschirpke. Error log processing for accurate failure prediction. In *WASL*, 2008.

[65] G. Salton and M. J. McGill. Introduction to modern information retrieval. 1986.

[66] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan, and P. Martin. Assisting developers of big data analytics applications when deploying on hadoop clouds. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 402–411. IEEE Press, 2013.

[67] J. W. Sheppard and W. R. Simpson. Improving the accuracy of diagnostics provided by fault dictionaries. In *Vlsi test symposium, 1996., proceedings of 14th*, pages 180–185. IEEE, 1996.

[68] M. Shepperd. A critique of cyclomatic complexity as a software metric. *Softw. Eng. J.*, 3(2):30–36, Mar. 1988.

[69] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering*, 39(4):552–569, 2013.

[70] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu. A general software defect-proneness prediction framework. *IEEE Transactions on Software Engineering*, 37(3):356–370, 2011.

[71] J. Stearley. Towards informatic analysis of syslogs. In *Cluster Computing, 2004 IEEE International Conference on*, pages 309–318. IEEE, 2004.

[72] G. Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007(011), 2002.

[73] N. Tillmann and W. Schulte. Parameterized unit tests. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 253–262, New York, NY, USA, 2005. ACM.

[74] R. Vaarandi. A data clustering algorithm for mining patterns from event logs. In *IP Operations & Management, 2003.(IPOM 2003). 3rd IEEE Workshop on*, pages 119–126. IEEE, 2003.

[75] R. Vaarandi. Simple event correlator for real-time security log monitoring. *Hakin9 Magazine*, 1(6):28–39, 2006.

[76] M. van der Bijl, A. Rensink, and J. Tretmans. Compositional testing with ioco. In A. Petrenko and A. Ulrich, editors, *Formal Approaches to Software Testing*, pages 86–100, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[77] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[78] J. A. Whittaker. What is software testing? and why is it so hard? *IEEE software*, 17(1):70–79, 2000.

[79] H. Wietgrefe, K.-D. Tuchs, K. Jobmann, G. Carls, P. Fröhlich, W. Nejdl, and S. Steinfeld. Using neural networks for alarm correlation in cellular phone networks. In *International Workshop on Applications of Neural Networks to Telecommunications (IWANNT)*, pages 248–255. Citeseer, 1997.

[80] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[81] F. Wotawa. Fault localization based on dynamic slicing and hitting-set computation. In *Quality Software (QSIC), 2010 10th International Conference on*, pages 161–170. IEEE, 2010.

[82] W. Xu, L. Huang, A. Fox, D. A. Patterson, and M. I. Jordan. Mining console logs for large-scale system problem detection. *SysML*, 8:4–4, 2008.

[83] J. Xuan, H. Jiang, Z. Ren, and W. Zou. Developer prioritization in bug repositories. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 25–35, Piscataway, NJ, USA, 2012. IEEE Press.

[84] K. Yamanishi and Y. Maruyama. Dynamic syslog mining for network failure monitoring. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 499–508. ACM, 2005.

[85] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10. ACM, 2002.

[86] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[87] T. Zimmermann and A. Zeller. Visualizing memory graphs. In *Software Visualization*, pages 191–204. Springer, 2002.