Mining Historical Test Logs to Predict Bugs and Localize Faults in the Test Logs

Anunay Amar and Peter C. Rigby
Department of Computer Science and Software Engineering
Concordia University
Montréal, Canada
an_mar@encs.concordia.ca, peter.rigby@concordia.ca

Accepted to ICSE'19. Camera-ready version will be available in February.

***Abstract*—Software testing is an integral part of modern software development. However, test runs can 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 paper we present techniques with the goal of capturing the maximum number of product faults, while flagging the minimum number of log lines for inspection.**

We observe that the location of a fault in a log should be contained in the lines of a failing test log. In contrast, a passing test 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 an 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, LogFaultFlagger finds 89% of the total faults and flags less than 1% of the total failed log lines for inspection. LogFaultFlagger drastically outperforms the previous work CAM. We implemented LogFaultFlagger as a tool at Ericsson where it presents daily fault prediction summaries to base station testers.

I. Introduction

Large complex software systems have 1000's of test runs each day leading to 10's of thousands of test log lines [17], [20], [34]. Test cases fail primarily due to two reasons during software testing: a fault in the product code or issues pertaining to the test environment [52]. 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 [34]. At Microsoft, techniques have been developed to automatically classify and ignore false test alarms [17]. At Huawei researchers have classified test failures into multiple categories including product vs environmental failure to facilitate fault identification [20].

In this work we focus on the Ericsson teams that are responsible for testing cellular base station 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 [49].

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 flagged for investigation, *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 and the results were presented in the technical track of ICSE'17 [20]. 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 50% 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 results in a significant proportion of environmental test failures relative to the number of product test failures. Due to the test environment, entire teams of testers exclusively analyze log test failures each day examining noisy failures to ensure that all product faults are found. To deal with this skewed data, 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 be flagged as product fault. SKEWCAM, which replaces KNN with *EKNN*, finds 89% 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 almost 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 85% 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.* LOGFAULTFLAGGER*: 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. LOGFAULTFLAGGER, identifies 89% of *FaultsFound* while only flagging 0.4% of the log lines. LOGFAULTFLAGGER finds the same number of faults as SKEWCAM, but flags less than 1% of the log lines compared to SKEWCAM's 28%.

This paper is structured as follows. In Section II, we provide the background on the Ericsson test process and the data that we analyze. In Section III, we detail our log abstraction, cleaning, *DiffWithPass*, and classification methodologies. In Section IV, we describe our evaluation setup. In Sections IV to VIII, we provide the results for our four log prediction and line flagging approaches. In Section IX, we contrast the approaches based on the number of *FaultsFound* and *LogLinesFlagged* for inspection, discuss performance and storage requirements, and describe how we implemented LOGFAULTFLAGGER as tool for Ericsson testers. In Section XI, we position our work in the context of the existing literature. In Section XII, we conclude the paper and describe our research contributions.

II. 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 paper, 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 1, we illustrate the integration testing process 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. In this work we study a six month period with 100's of thousands of test runs and associated test logs.[1]

III. 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 set difference between the failing log and the last passing log to remove lines that are not related to the failure (*i.e. DiffWithPass*). 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.

A. 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 indicates 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.

[1]Ericsson requested that we not report specific test and log numbers

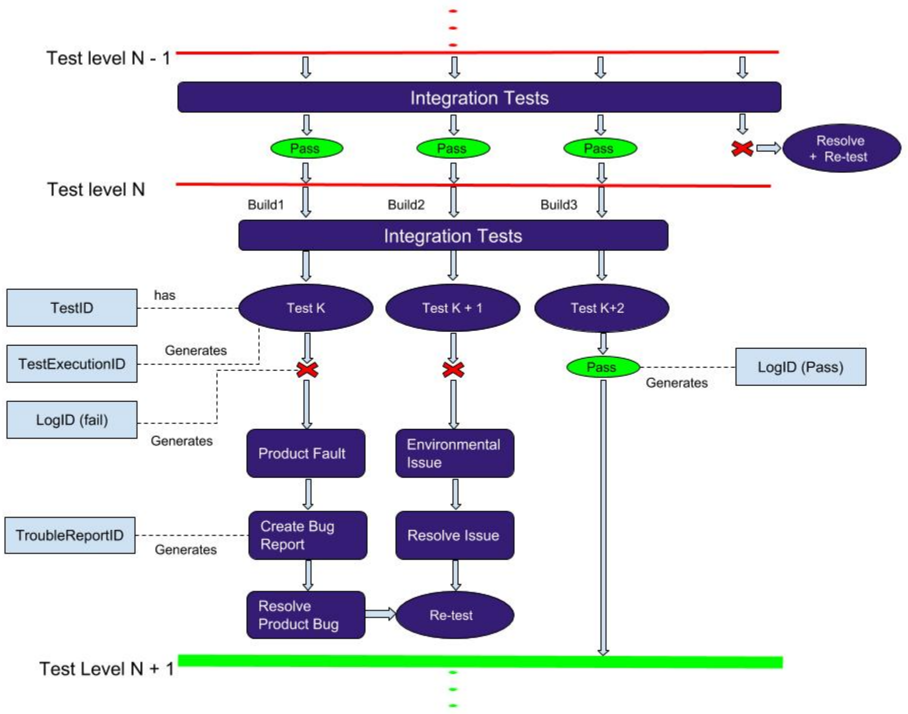
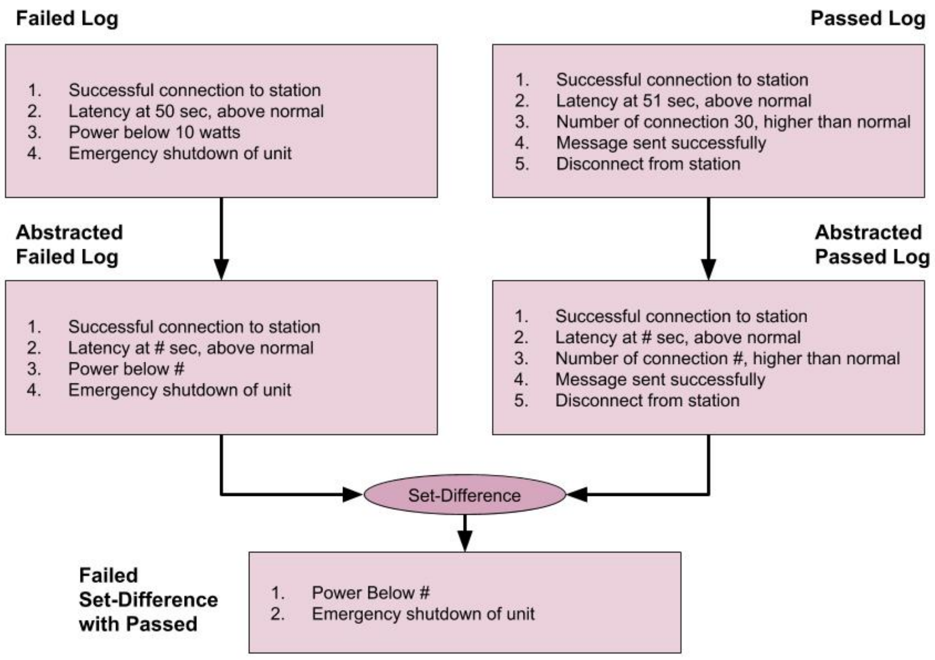


Fig. 1: The Ericsson integration test process. Code has already gone through earlier developer testing stages (N-1) and will continue to later integration stages (N+1). The data we extract is shown in the square boxes, *e.g.,* LogID.



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
```

Fig. 2: Processing stages: First, the logs are abstracted. Second a set-difference operation is performed between the passing and failing log (*DiffWithPass*). Third, only the lines present in the failing log are stored.

By removing this information, abstract lines contain the essence of each line without the noisy details.

For example, in Figure 2, 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 build upon Shang *et al.*'s [44] log abstraction technique modifying it for test logs.

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.

B. 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 an approach where we remove the lines that occur in the passing log from the failing log. In our example, in Figure 2, 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 the *DiffWithPass* operation with all the previous passing logs is computationally expensive and grows with the number of test runs, $O(n)$. For each failure we have to compare with the test's previous passing runs, which would lead to over 455 million comparisons across our dataset. The number of passes makes this impractical. To make our approach scalable, we note that a passing log represents an acceptable state for the system. *We perform a set difference of the current failing log with the last passing log.* Computationally, we perform one *DiffWithPass* comparison, $O(1)$. This approach reduces the number of noisy lines in a log and as we discuss later reduces the storage and computational requirements.

C. 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 operationalized 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 in a log that has 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 3, 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.

We further stress that the individual log lines are **not** marked by developers as being related to a fault or bug report. While this data would be desirable, we have not come across it on industrial projects. Instead, as can be seen in Figures 1 and 3 the failing build and associated test failure are linked to a bug report. After performing abstraction and *DiffWithPass* we store and increment the failure and fault count for each line in the log for later IR processing to determine which log lines have high predictive power.

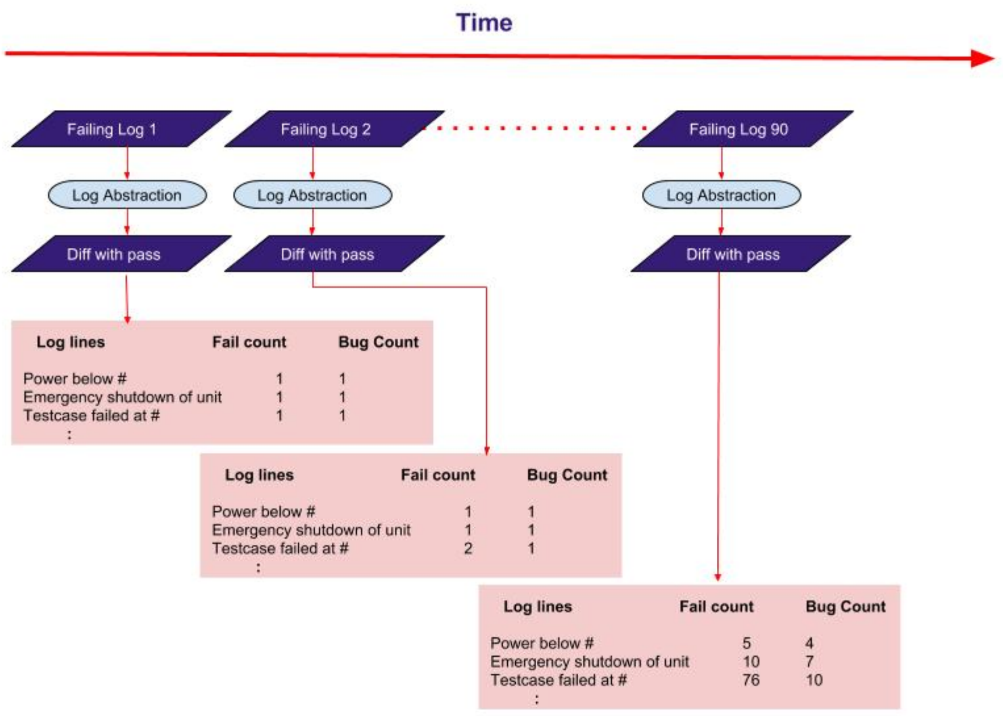
D. TF-IDF and line-IDF

Identifying faults based on test failures and bug reports is too simplistic. Term Frequency by Inverse Document Frequency (TF-IDF) is used to calculate the importance of a term to a document in a collection [43]. 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 a log “document” d, N denotes the total number of logs for a test, and N_t denotes the number of logs for a test that contains the term t [43] [20].

We have discussed in earlier sections that rare log lines should be strong indicators of faults. We use IDF (Inverse



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
:		

Fig. 3: 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. For example "Power below #" has occurred in 5 fails and 4 times a bug has been reported.

document frequency) to operationalize the importance of a log line to a test log. line-IDF is defined as:

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

Where N denotes the number of logs for a test, and N_l denotes the number of logs for a test that contains the log line l.

E. 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.

Many techniques exist to assign values to the features. We use three techniques. For CAM and SkewCAM, were the features are the terms in a log, we use the standard TF-IDF formula (see Equation 1). For LogLiner, were the feature is a line, we use use line-IDF (see Equation 2). For LogFaultFlagger we multiple the fault frequency of a line by the line-IDF (see Equation 7).

F. Cosine Similarity

To find similar logs and log lines to predict faults we use cosine similarity. It is defined as [43] [40]:

$$similarity = \cos\theta = \frac{\vec{L_1} \cdot \vec{L_2}}{\|L_1\|_2 \|L_2\|_2} \tag{3}$$

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 log lines are the most likely indication for the fault.

*G. 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.

IV. Evaluation Setup

Ericsson testers evaluate test failures on a daily basis. As a result, we run our simulation on a daily basis training on all the previous days. This simple incremental simulation framework has been commonly been used in the research literature [3], [17], [20], [55]. Our simulation period runs for 6 months and covers 100's of thousands of test runs and logs. 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 $D = t$. We repeat this training and testing cycle for each nightly run until we reach $D = T$.

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

$$FaultsFound = \frac{TotalCorrectlyPredictedFaults}{TotalTesterReportedFaults} * 100 \quad (4)$$

$$LogLinesFlagged = \frac{TotalFailedLogLinesFlagged}{TotalLogLinesAllFailedLogs} * 100 \quad (5)$$

V. Result 1. CAM: TF-IDF & KNN

CAM has successfully been used at Huawei to categorize test logs [20]. We re-implement their technique and perform a replication on Ericsson test logs. We discussed the data processing steps in Section III. 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 $K = 15$ [20], we vary the number of neighbours from $K = 1$ to 120.

Table I 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 $K = 15$ CAM finds 50% of the product faults. However, when we increase K to 30 it only captures 48% of the product faults.

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, CAM performs word based TF-IDF which generates large term-based 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. CAM is computationally expensive.

TABLE I: CAM: TF-IDF & KNN

K	% FaultCaught	% LogLineFlagged	Execution Time (mins)
1	47.30	4.13	420
15	**50.00**	**4.38**	**444**
30	47.23	4.36	458
60	47.14	4.07	481
120	47.43	4.23	494

TABLE II: SkewCAM: CAM with *EKNN*

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

TABLE III: LogLiner: Line-IDF & *EKNN*

K	N	% FaultCaught	% LogLineFlagged	Execution Time (mins)
1	1	47.23	0.06	30
15	1	67.48	0.14	46
30	1	68.22	0.16	52
60	1	68.22	0.17	68
120	1	68.22	0.17	91
1	10	47.27	0.56	39
15	10	82.35	2.39	85
30	**10**	**84.60**	**2.98**	**90**
60	10	86.05	3.92	98
120	10	86.05	4.26	127

VI. 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 III-G, 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.

TABLE IV: LogFaultFlagger: PastFaults $*$ Line-iDF & EKNN

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

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 [20] 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 II shows that more neighbours catch more product faults but also flag 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 increases to only 91%, but the lines flagged increases to 44%.

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 lines for inspection. Testers must now face the prospect of investigating entire log files.

Despite removing the threshold calculation, SkewCAM is still computationally expensive because like CAM it applies term-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.

VII. 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 a new 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 III.

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, which is a reworking of Equation 2:

$$IDF(Line) = \log \frac{TotalNumLogs}{LineInLogsCnt} \tag{6}$$

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 and 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 III shows the impact of changing these parameters. Low parameter values $N = 1$ and $K = 1$ lead to *FaultsFound* at 47% with < 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 $N =$ "all the lines in a log," LogLiner finds 88% of the faults, but flags 29% of the lines, a similar result to SkewCAM (not shown in a figure).

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.

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

VIII. Result 4. LogFaultFlagger: PastFaults $*$ Line-IDF & EKNN

LogLiner flags fewer lines, but drops slightly in the number of *FaultsFound*. We build on LogLiner with LogFaultFlagger 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 line fault frequency (FF) according to the following equation:

$$\begin{aligned} \text{FF-IDF}(Line) &= (LineFaultCount + 1) * IDF(Line) \\ &= (LineFaultCount + 1) * log \frac{TotalNumLogs}{LineInLogsCnt} \end{aligned} \tag{7}$$

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 IV 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

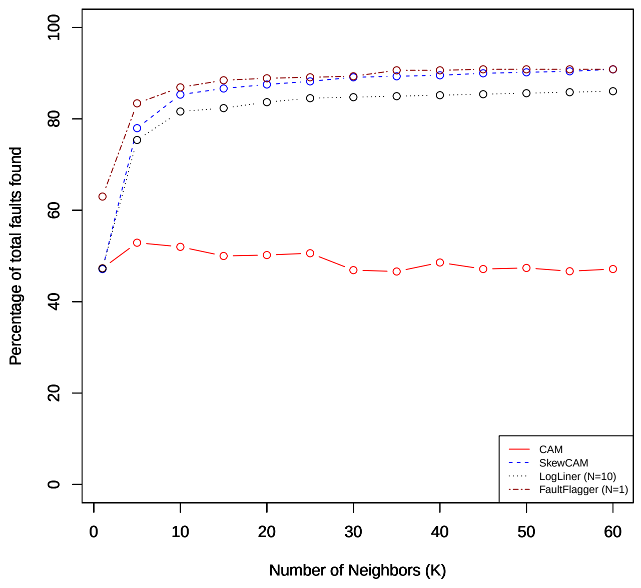


Fig. 4: *FaultsFound* with varying K. SKEWCAM and LOGFAULTFLAGGER find a similar number of faults. CAM finds 50% or less of the total faults.

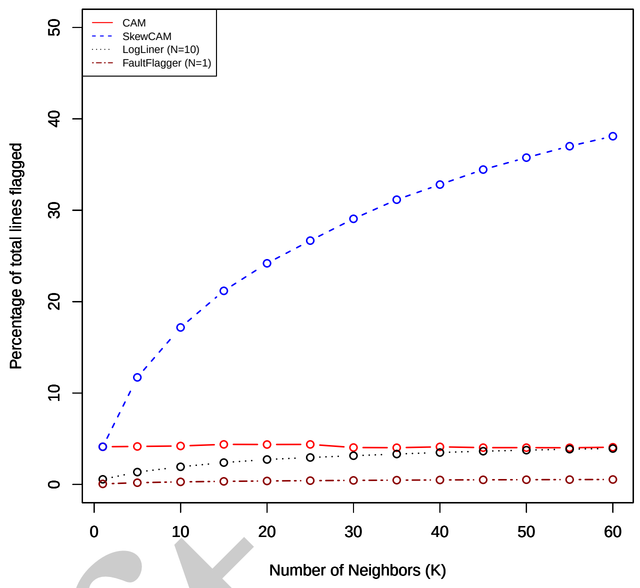


Fig. 5: *LogLinesFlagged* with varying K. SKEWCAM flags an increasing number of lines, while LOGFAULTFLAGGER remains constant around 1% of total log lines. CAM flags relatively few total log lines because it predicts fewer faults only finding 50% of the total faults.

further comparisons and find that LOGFAULTFLAGGER finds 89% of the total faults with 0.4% of total log lines flagged for inspection.

Compared to SKEWCAM, LOGFAULTFLAGGER finds the same number of faults, but SKEWCAM flags 28% of total log lines compared LOGFAULTFLAGGER < 1%. Compared to LOGLINER, LOGFAULTFLAGGER finds 4 percentage points more faults with 2.5 percentage points fewer lines flagged.

LOGFAULTFLAGGER finds 89% of the total faults and flags only 0.4% of lines for inspection.

IX. Discussion

Testers want to catch a maximal number of faults while investigating as few log lines as possible. We discuss the reasons why the techniques differ in the number of correctly identified test failures that lead to faults, *FaultsFound* in Figure 4, and the number log lines used to make the prediction, *i.e.* the lines that are flagged for manual investigation, *LogLinesFlagged*, in Figure 5. The figures also provide a visual representation of the impact of changing the number of K neighbours. We also discuss the performance and storage requirements and the implementation of the best approach, LOGFAULTFLAGGER, as a tool for Ericsson testers.

CAM technique: We re-implemented Huawei's CAM [20] technique and evaluated it on a new dataset. CAM uses simple term based TF-IDF to represent failed test logs 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, although the CAM tool provides a display option to diff the failing log, it uses the entire log in its prediction and so CAM does not flag individual log lines that are the likely cause of the fault. Instead it only categorizes test failures into, for example, product vs environmental failure. The second limitation is that CAM performs poorly on the Ericsson dataset, see Figure 4 and 5. 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 is 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 4 shows that SKEWCAM plateaus finding 89% of the product faults solving the first limitation of CAM. SKEWCAM's major limitation is that it flags an increasingly large number of log lines in making its fault predictions. Figure 5 shows that as the number of K neighbours increases so too does the number of *LogLinesFlagged*. As a result, testers must manually examine many log lines to identify the cause of the failure. 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 *LogLines-*

Flagged, we introduce a new technique called LogLiner. Instead of using terms as the unit of prediction, LogLiner modifies TF-IDF by employing IDF at the log line level. The line-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 the 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. LogLiner plateaus at identifying 85% of the faults, while flagging 3% of the lines. *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.*

LogFaultFlagger technique: To improve the number of *FaultsFound* and reduce the number of *LogLinesFlagged*, we suggest a new technique called LogFaultFlagger that uses the association between log lines and *LineFaultCount*. LogFaultFlagger 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 4 and 5 show that LogFaultFlagger plateaus finding 89% of the faults while consistently flagging less than 1% of the of the total log lines for investigation. As the figures show, in order for SkewCAM to find the same number of faults as LogFaultFlagger it must flag an increasing and drastically larger number of lines for inspection.

LogFaultFlagger not only outperforms the state-of-the-art in terms of effectiveness, it also introduces the use of log abstraction and *DiffWithPass* to test log processing which has substantial benefits in terms of performance and storage.

A. Performance and Log Storage

The last column of Table I, Table II, Table III, and Table IV show the execution time of CAM, SkewCAM, LogLiner, and LogFaultFlagger respectively. We can see that both CAM and SkewCAM are computationally more expensive than LogLiner and LogFaultFlagger. At $K = 30$, CAM, SkewCAM, LogLiner (N=10) and LogFaultFlagger (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 term 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 terms present in the failing test logs, and L denotes the total number of failing test logs. In contrast, LogLiner and LogFaultFlagger use line-IDF where the line is the feature unit, v, where $v \ll V$. As a result LogLiner and LogFaultFlagger have a time complexity of $O(|v| \cdot |L|)$, where v denotes the set of unique log lines in the set of failed logs.

Performing log analysis on huge log files is tedious and expensive. CAM, SkewCAM, LogLiner, and LogFaultFlagger all require 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 *DiffWithPass*. 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 *DiffWithPass* we were able to reduce the log storage size by 94%. This reduction drastically reduces the storage requirements and allows companies to store the important part of test logs for a longer time period.

B. Implementing the LogFaultFlagger tool at Ericsson

LogFaultFlagger was implemented as a tool at Ericsson. To reduce disruption and encourage adoption, a field was added to the existing testing web dashboard to indicate whether the test failure is predicted to lead to a product fault or an environmental failure. The tester can click to view the log in a *DiffWithPass* view that shows only those lines that are in the current failing log. While this view is still available, feedback from Ericsson testers indicated that they preferred to view the flagged lines in the context of the entire log. The view of the log was modified to highlight the flagged log lines and allows testers to jump to the next flagged line. Another product team at Ericsson hired one of our researchers to re-implement LogFaultFlagger in a new test setting. As we discuss in the threats to validity, a short tuning stage is required, but the overall technique is dependent only on storing historical test logs and does not depend on a particular log format or development process.

X. Threats to Validity

We report the results for a single case study involving 100's of thousands of test executions over a six month period. Since the test failure data is highly skewed because of the significant proportion of environmental failures, we use a large number of neighbours, $K = 30$. It is simple to adjust the value of K based on the number of faults that lead to bug reports for other projects. Indeed, the success of LogFaultFlagger has lead to its adoption on another Ericsson team. Although in the early stages, the initial results are promising and since there are fewer environmental failures the data is more balanced and standard KNN has replaced *EKNN*. We have also experimented with other models including, logistic regression, decision tress, and random forests. Although a complete discussion is out of the scope of this paper, we note that decision tress and random forests perform less well than simple logistic regression and KNN.

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 increases 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 LogFaultFlagger 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 faults 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.

XI. Related Work

A. Fault Location

There is large and successful body of work on locating faults within source code. Traditional fault location techniques use program logs, assertions, breakpoints in debuggers, and profilers [2], [6], [10], [42]. More advanced fault identification techniques use program slicing-based algorithm, program spectrum-based algorithm, statistics-based algorithm, program-state based algorithm, and machine learning-based algorithm [4], [5], [8], [9], [12], [22], [29], [38], [46], [51], [54]. In contrast, our algorithms LogFaultFlagger and LogLiner flag log lines that are likely related to faults in the test log not the source code. Testers can use the flagged log lines to determine the reason behind the test failure. Techniques to trace log lines back to test cases and ultimately to the part of the system under test are necessary future work so that our log line location technique can be automatically traced back to the faulty source code.

B. 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 use statistical regression models and machine learning models to predict faults in software modules [1], [7], [13], [24], [24], [25], [27], [28], [32], [33], [35], [39], [47]. Herzig [16] performed preliminary work combining 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. With the exception of Herzig [16] the bug models we are aware of do not include test failure information. In contrast, our model uses not only test outcomes, but also the dynamic information from the test logs to predict faults.

C. Log Analysis and Failure Clustering

Logs are an important part of operating a production system. The majority of log analysis work has focused on production logs of live systems or traces of running code. These works have used statistical learning approaches to identify sequences in logs [21], [36], [50], find repeating and anomalous patterns [14], [18], [31], [45], [53], and clustering similar logs [20], [30], [36], [37], [48]. We have adapted the log abstraction approaches to work on test logs [21]. Since we have an external indicator of success, *i.e.* a test pass or fail, we use *DiffWithPass* that reduces log storage size and helps testers in identifying the cause of the failure in a log.

D. Categorizing Test Failures

The testing literature is vast, ranging from test selection and prioritization [11], [15], [26], [56], [58] to mutation testing [19], [23], [57]. In this work, we focus on false alarms, *i.e.* non-product failures, that are common on large complex systems [17], [20], [34], [41]. These “false alarms” have received attention because successful classification of false test alarms saves time for testing teams. Throughout the paper we have contrasted and replicated the state-of-art on test log classification, CAM [20]. False alarms can also slow down the development team when test failures stop the build. For example, this issue was addressed at Microsoft by automatically detecting false test alarms [17]. Microsoft uses association rules to classify test failures based on configuration information and past pass or fail results. The classification does not consider the test logs. 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 the likely log line location of the failure.

XII. Concluding Remarks

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

First, using log abstraction, we are able to reduce the log storage requirement by 78%. We also observe 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. We perform a set-difference between the failing log and the last passing log. *DiffWithPass* further reduces the storage requirement to 94%. *DiffWithPass* also reduces the noise present in the failed test log helping testers isolation faults in the log.

Second, our discussions with testers revealed that they want to find the most faults while investigating the 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 [21]. 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 faults and locate suspicious log lines, we introduce an approach where we train our model on a subset of log lines that occur in current failing test log. LogFaultFlagger identifies the rarest lines that have lead to past faults, *i.e.* PastFaults * Line-IDF + *EKNN*. In our Ericsson tool, LogFaultFlagger highlights the flagged lines in the log for further investigation by testers.

Third, LogFaultFlagger drastically outperforms the state-of-the-art, CAM [20]. 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 LogFaultFlagger, however, it flags 28% of the log lines compared to the less than 1% flagged by LogFaultFlagger.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

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

[15] H. Hemmati. Advances in techniques for test prioritization. Advances in Computers. Elsevier, 2018.

[16] 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.

[17] 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.

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

[19] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, Sept 2011.

[20] 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.

[21] 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.

[22] 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.

[23] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 654–665, New York, NY, USA, 2014. ACM.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

[38] 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.

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

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

[41] M. T. Rahman and P. C. Rigby. The impact of failing, flaky, and high failure tests on the number of crash reports associated with firefox builds. In *Proceedings of the 2018 Foundations of Software Engineering (Industry Track)*, ESEC/FSE 2018. ACM, 2018.

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

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

[44] 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.

[45] 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.

[46] 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.

[47] 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.

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

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

[50] 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.

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

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

[53] 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.

[54] 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.

[55] 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.

[56] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[57] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang. Predictive mutation testing. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[58] Y. Zhu, E. Shihab, and R. PC. Test re-prioritization in continuous testing environments. In *2018 IEEE International Conference on Software Maintenance and Evolution*, page 10, 2018.