Studying Test Case Failure Prediction for Test Case Prioritization

Tanzeem Bin Noor
Highjump Software Inc.
125 Commerce Valley Drive West
Markham, Ontario L3T 7W4
tanzeem.noor@gmail.com

Hadi Hemmati
Department of Electrical and Computer Engineering
2500 University Drive NW
Calgary, Alberta T2N 1N4
hadi.hemmati@ucalgary.ca

ABSTRACT

Background: Test case prioritization refers to the process of ranking test cases within a test suite for execution. The goal is ranking fault revealing test cases higher so that in case of limited budget one only executes the top ranked tests and still detects as many bugs as possible. Since the actual fault detection ability of test cases is unknown before execution, heuristics such as “code coverage" of the test cases are used for ranking test cases. Other test quality metrics such as “coverage of the changed parts of the code" and “number of fails in the past"’ have also been studied in the literature. **Aims:** In this paper, we propose using a logistic regression model to predict the failing test cases in the current release based on a set of test quality metrics. **Method:** We have studied the effect of including our newly proposed quality metric (“similarity-based" metric) into this model for tests prioritization. **Results:** The results of our experiments on five open source systems show that none of the individual quality metrics of our study outperforms the others in all the projects. **Conclusions:** However, the ranks given by the regression model are more consistent in prioritizing fault revealing test cases in the current release.

KEYWORDS

Test Case Prioritization; Test Case Quality Metrics; Regression Model

ACM Reference format:
Tanzeem Bin Noor and Hadi Hemmati. 2017. Studying Test Case Failure Prediction for Test Case Prioritization. In *Proceedings of ACM International Conference on Predictive Models and Data Analytics in Software Engineering, Toronto, Canada, November 2017 (PROMISE'17),* 10 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In software testing, the ultimate effectiveness of test cases is measured based on their ability to detect real faults, i.e., how many faults the test can reveal when it executes. Unfortunately, this measure is not always practical because one needs to know about the effectiveness of test cases before execution (e.g., in the context of test case prioritization).

PROMISE'17, November 2017, Toronto, Canada

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Test case prioritization is an important element in software quality assurance in practice. It is even more crucial in modern software development processes, where continuous integration is widely adapted. Source codes are frequently changed due to either adding new functionality or fixing known bugs. A study shows that in Google, more than 20 code changes occur in every minute [37]. Each change typically requires (re)testing the software. This will typically lead to huge test suites. For instance in the previous example from Google, the frequent changes has led to 1.5 million test executions per day at Google, in 2014.

Most of the time, software companies have limited resources (e.g., personnel, budget, and time) for testing and they can not retest the entire test suite after every change. To solve this problem, test case prioritization is used, which assigns ranks to the test cases and the tests are executed according to the order of their ranks, until the testing budget is ran out. Test cases are prioritized in a way that they detect the potential faults earlier. Therefore, it is very important to rank the effective test cases higher so that the faults can be detected by executing only the higher ranked tests within specific budget. As the ultimate effectiveness metric (i.e., actual fault detection rate) can be measured only after test execution, it can not be used in the context of test case prioritization, which assigns ranks to the test cases before execution. Therefore, we need other test quality metrics to estimate the effectiveness of the tests cases and prioritize them accordingly.

Several metrics such as code coverage, code change-related metrics, previous fault detection capability etc. are used to measure the test case quality. Code coverage measures how much of the source code (e.g., number of lines, blocks, conditions etc.) from the program is covered during the test execution. In contrast, change-related test quality metrics emphasize more on covering the changed part of the program source, which is also very crucial, specially in the context of regression testing. Previous fault detection metrics take a different approach to measure test quality. They consider a test as effective if it detects any fault from previous releases.

In this paper, we have studied the effectiveness of several existing metrics (i.e., coverage, change-related metrics, previous fault detection capability etc.) in prioritizing fault revealing tests. We specifically focus on combining existing metrics using a regression model and predict the probability of a test case detecting a fault in the current release. Therefore, test cases are prioritized based on their probability of failure (the higher the probability, the higher the priority). We have conducted a series of empirical study using five open source java projects (total 247 versions) to evaluate ranking of the failing test cases provided by the regression model in the context of test case prioritization. The results of our study show that in general, combining several existing metrics in a regression

model provides better ranking compared to the ranking provided by individual metrics.

We have also experimented with including a newly introduced test quality metric "Similarity-based metric" [25] into our regression models. The results show that the model including the new metric outperforms the model with only traditional metrics.

The rest of this paper is organized as follows: We mention background and some basic quality metrics in section 2. Our proposed regression model using similarity-based metrics and existing metrics is explained in section 3. We discuss our experiments and results in section 4. Section 5 mentions some of the related works. Finally, section 6 concludes the paper and mentions our future work.

2 BACKGROUND

Test cases are prioritized using some quality metrics. Test case quality metrics are used to evaluate the tests and find the scope of improvement required in the test cases. A well-known set of such quality metrics are test adequacy criteria [26, 28]. A test adequacy criterion is a predicate that defines which properties of a program must be exercised, if the test is to be considered adequate with respect to the specific criterion [13]. Two main types of test adequacy criteria are coverage-based and fault-based test adequacy criteria [26, 28].

Besides these test adequacy criteria, there are some other quality metrics that may have high correlation to fault detection ability of the test cases, such as historical fault detection, coverage, size of the test, complexity, code churn, etc. [6]. In the rest of this section, we explain two categories of test adequacy criteria and historical fault detection metric.

2.1 Coverage-based Test Adequacy Criteria

Coverage is a commonly used test adequacy criteria that indicates how much of the program is executed when the test case runs. A test case may reveal a fault when it executes a segment from the program that causes the failure and therefore, high coverage (executing more segments from the source code) of a test case may indicate higher probability of failure. Coverage based test adequacy criteria is further grouped into control-flow coverage and data-flow coverage. Adequacy criteria based on control-flow could be categorized into number of coverage levels, such as Procedure/Method Coverage, Branch Coverage, Statement Coverage, Path Coverage and condition coverage.

Data-flow coverage criteria depend on the flow of data through the program. These criteria focus on the values associated with variables (def) and how these associations can affect the execution of the program (use) [38]. The most well-known data-flow coverage criterion is DU-pair coverage, where the goal is covering all pairs of definitions and uses per variable [28]. According to data-flow coverage criterion, a test case is considered to be effective when it's DU-pair coverage is high.

2.2 Fault-based Test Adequacy Criteria

Fault-based adequacy criteria measures the quality of the test cases by their ability to detect known faults, as an estimate for their ability for detecting unknown faults [17]. The known faults are either real faults or artificial faults (mutants – small syntactic changes in the program [28]). In mutation testing, the effectiveness of the test case is measured by the number of killed mutants by the test case. More killed mutants by a test case infers that the test case is more effective in finding real faults in the program, as well and hence the tests are ranked higher than the others.

2.3 Historical Fault Detection

Historical fault detection quality metric considers a test case to be effective in the current release if the same test was also able to detect faults in previous releases (i.e., the test has failed previously) [6, 20, 39]. The rationale behind this is that studies have shown that the tests with higher historical fault detection rate are more likely to fail in the current version as well [39]. Therefore, the previously failing tests are also prioritized in the current release, specially, in the context of regression testing.

However, the test case that fails in the current release might not be always same as the previously failed test cases, but they might be quite similar. For example, when a test case is slightly modified from previous release, the test case is not considered as effective according to the traditional historical fault detection quality metric, since it did not exist in the previous releases, to fail and therefore, it is ranked lower in the prioritized list. However, if the original test case from the previous release was a failing test, there is still high chance that the modified version fails in the new release. In our previous work [25], we have shown such test cases that are similar to the previously failing test cases, in terms of their method sequence calls.

We proposed a set of similarity-based metrics that uses execution traces of the previously failed test cases from history and measures quality of a test case in the current release using it's similarity to the past failing traces. We found that our proposed similarity-based metrics which also utilize historical faults (i.e., failing traces), outperforms the traditional historical fault detection metric, in terms of identifying fault revealing test cases in the current release.

3 METHODOLOGY

In this section, we investigate the effectiveness of individual test quality metrics as well as a combined version of them in a regression model. In addition, we have proposed to build a regression model that uses the existing quality metrics and our recently proposed similarity-based metrics. In the rest of this section, we explain the quality metrics that we use to build the model in this study.

3.1 Traditional Quality Metrics

In this paper, we consider the following traditional quality metrics for test case prioritization:

3.1.1 Traditional Historical Fault detection metric (TM). The traditional historical fault-detection metric (TM) considers a test as effective if it detects any fault from previous versions (i.e., failed previously) [1, 6, 20, 39]. We measure TM of a a given test case, in one release, by counting the number of it's failing occurrences. The higher the TM, the higher the test case rank.

3.1.2 Method Coverage (MC). Method Coverage (MC) is a simple control-flow coverage criterion that measures procedure/method coverage of a test case. For each test case, the method coverage

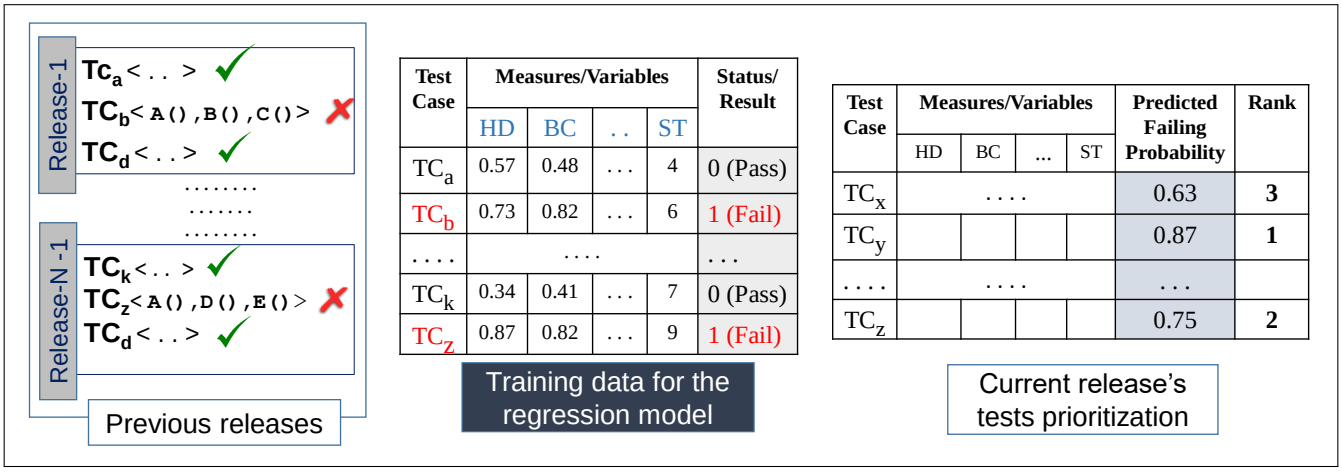

Figure 1: Test case prioritization using regression model

refers to the number of source code methods called from the test case (directly or indirectly), divided by the total number of methods in the source code [26]. According to MC metrics, the test case that covers more methods/procedures, is considered to be more effective and ranked higher.

3.1.3 Changed Method Coverage (CMC). Often times in test prioritization, specially in the context of regression testing, one would like to make sure that modified parts of the source code are well-tested [6]. The simple MC coverage is not the best metric for measuring such coverage because for instance one can cover most of the modified part of the source code with a low-to-moderate total coverage (MC) [25]. In this paper, we define Changed Method Coverage (CMC) metric of a test case as the number of changed methods from the previous version that are directly or indirectly executed when the test case is executed, and again the higher CMC value of a test case suggests a higher rank.

3.1.4 Size of Tests. Typically, size of a test case refers to either the LOC (Line of Codes) or the number of "assertions" in a test case. Note that size of a test case is not necessarily correlated with its coverage [28]. For example, test case t_1 can cover a method with one assertion and test case t_2 do the same but try it with 5 more assertions. In this case, t_2 may have higher chances to detect faults than t_1 [25]. In our study, we consider LOC (Line of Codes) as a measure of test size, where larger sizes indicate that the tests have higher probability to detect faults.

3.2 Similarity-based Quality Metrics

In our previous work, we defined a family of similarity-based metrics that prioritize the test cases when they are similar to any of the failed test cases from previous versions [25]. We defined similarity between test cases based on their execution traces containing sequences of method calls from program source code. Execution traces of method calls for all previously failed test cases are collected from history (once per release) and compared against the method call sequences of the current release test cases.

A similarity function was used to determine the current release test's similarity to the previously failed test cases. The inputs of a similarity function are: a) traces from previously failed test cases and b) a trace from the current release test case. The similarity function, at the end, returns a similarity value as output. Finally, the quality of the test cases are measured based on the returned similarity values and those with higher similarity values are prioritized. [25].

In our previous work, we defined four different similarity functions to implement the similarity-based metrics [25]. In this study, we propose to build a regression model using the similarity-based metrics along with the existing metrics mentioned in the previous subsection. In the rest of this subsection we briefly explain the four similarity-based metrics:

3.2.1 Basic Counting (BC). The Basic Counting (BC) similarity function simply counts the unique method calls from the current release's tests trace that also appear in the past failing sequences. In other words, BC is equal to the number of common and unique method calls between the given test case trace and the previously failed traces. We have normalized the similarity values between 0 and 1 by diving the total number of common and unique method calls by the total number of unique method calls from the history [25].

3.2.2 Edit Distance (ED). The general edit distance function is a sequence-aware function that considers the order of method calls in the traces. We have used one of the most well-known implementation of edit-distance, called Levenshtein distance [8], where the similarity between two sequences of method calls are calculated using the different rewarding points that are assigned to all the matches, mismatches and gaps in the sequences [25]. So, the lower edit distance indicates higher similarity and the test cases having higher similarity are ranked higher.

3.2.3 Hamming Distance (HD). Hamming Distance between two sequences is the minimum number of edit operations (insertions, deletions, and substitutions) required to transform the first sequence into the second [8, 15]. Note that hamming is only applicable on

Table 1: Projects Under Study

Projects	#Faults (#Versions)	#Test Cases	Median number of Test cases per version
JFreeChart	26	2,205	1,751
Closure Compiler	133	7,927	7,066
Commons Math	106	3,602	1,976
Commons Lang	65	2,245	1,757
Joda Time	27	4,130	3,748

identical length inputs and is equal to the number of substitutions required in one input to become the second one [8]. However, in our case, the sequences of method calls may have different lengths. Therefore, to force them to have an identical length, at first, we have made a vector V that contains all the method calls from two comparing sequences. Then, we have encoded the each test case using a binary vector of the length of V, where a bit is true if the corresponding method call appears in the test case. Finally, the hamming distance between two sequences of method calls is measured by applying XOR operation between their binary encoded representations and then normalized between 0 and 1 by dividing the sum of XOR values by the length of V [25]. The low hamming distance value of a test case means the test case has high similarity with the previous failing test cases and should be ranked higher.

3.2.4 Improved Basic Counting (IBC). We proposed an Improved Basic Counting (IBC) similarity function that combines the Basic Counting (BC) and Hamming Distance (HD) similarity functions. IBC is a weighted version of BC and HD, where BC values are used as the default similarity value of IBC and therefore, it is called the improved BC. However, if the BC similarities are very low (i.e., less than a threshold) then the IBC will use the HD similarity value (unless HD's values are also lower than the threshold, which in this case the default BC value would be used) [25]. Therefore, test cases having higher IBC values will be prioritized.

3.3 Building Regression Model

In this paper, we propose to combine our similarity-based metrics with existing quality metrics into a regression model and prioritize the test cases based on the estimated failure probabilities. But first we build a regression model using only the traditional metrics (ED, HD, BC, IBC, TM, MC, CMC, ST) of each test case from previous releases as independent variables. Basically, to predict the ranks of the test cases in the current version of a project, the regression model is trained using different quality measures from all the previous versions. For example, to rank the test cases in a latest version (e.g., version 10) of a project, the model is trained using all quality measure values from version 1 to 9.

Since a test case can either pass or fail in the actual execution, the dependent variable of the regression model is binary (pass or fail), which makes the model a logistic regression model. Now given these quality measures of each test case from previous releases, the model is actually trained based on the actual status of each test case (i.e., whether it was passed or failed).

After applying the trained model on the current release's test cases, they are ranked based on the probability value of the test being failed. The overall process of combining quality metrics using a regression model is shown in Figure 1, where the higher rank of test case TC_y indicates that it should be executed earlier than other test cases (e.g., TC_z, TC_x) to reveal faults in the current release.

It is worth mentioning that there might be several independent variables in the model that are highly correlated, which is referred as *multicollinearity* problem. In addition, all the variables might not be statistically significant for the prediction model. To deal with these problems, we have removed highly correlated variables (i.e., any variables with correlation higher than 0.5) as suggested by Shihab *et al.* [34]. We have performed this removal in an iterative manner, until all the factors left in the model has a Variance Inflation Factor (VIF) below 2.5, as recommended by previous works [4, 34]. Thus, the regression model is built with the least number of uncorrelated factors. In addition, we have also measured the p-value of the independent variables for statistical significance and ensured that it is less than 0.05. Note that the remaining number of uncorrelated factors in the model might be different for different versions of a project [34].

So far we only used the traditional metrics for prediction, but the regression model can also contain our similarity-based metrics. We will discuss this in RQ2 of the Empirical Study section.

4 EMPIRICAL STUDY

In this section, we explain our empirical study and discuss the results of our experiments.

4.1 Research Questions

We have investigated the following two research questions in this study:

RQ1: Can combining the traditional test quality metrics using a regression model improve test prioritization compared to ranking the test cases using the individual metrics?

RQ2: Can we improve test prioritization results by adding similarity-based quality metrics into the traditional model of RQ1?

4.2 Subjects Under Study

We have used five different Java projects in our experiment. The projects have been retrieved from the *defects4j* database [7]. The database provides 357 faults and 20,109 Junit tests from five different open-source Java projects as mentioned in Table 1 [25]. All the faults are real, reproducible and have been isolated in different versions [18]. There is a fixed version and a faulty version for each fault of the program source code. The faulty source code is modified to remove the fault in the fixed version. The test cases are exactly the same in both of the faulty and the fixed versions. In each version, there is at least one test case (a Junit test method) that fails on the faulty version but passes on the fixed version [24] [25].

Table 2: Number of Studied Version

Projects	#Studied Versions	#Versions (# new/modified tests >=5)	#Versions (TM works)
Commons Lang	60	41	20
JFreeChart	24	16	2
Commons Math	43	36	12
Joda Time	25	16	0
Closure Compiler	95	85	0

4.3 Data Collection

We have extracted the basic quality metrics, i.e., Size of Testcase (ST), Method Coverage (MC), Changed Method Coverage (CMC) and Traditional historical fault-based Metric (TM) directly from the projects.

ST is the number of uncommented statements in a test method that has been derived from Java Abstract Syntax Tree (AST) parser using Eclipse JDT API [10]. MC of a test case is the number of unique methods called during the test execution. We have calculated this measure from the execution traces produced by Daikon and AspectJ [25]. CMC of a test case is the number of unique method calls that are called by the test case and have been changed since the previous version. To calculate CMC, we first identified the list of source code method names in the current version that have been changed from the immediate previous version. Then we count the change methods that also appear in the execution trace of modified/new test cases of the current version. The final CMC value of a test case has been normalized between 0 and 1 by dividing the total number of changed methods by the total number of unique method calls from the test execution trace.

TM measures how many times the tests from the current release failed previously. However, in our study, we found that in two of the projects (i.e., Commons Math and Closure Compiler), TM is not available in any version, as the fault revealing test cases in any given release did not fail in the previous releases. Therefore, test cases could not be ranked using TM in these two projects and in JFreeChart project, TM only works for 2 versions. Table 2 shows the detailed number of versions where TM works for each project .

Other than those 4 basic quality metrics, we have also extracted the four similarity-based metrics (BC, HD, ED, IBC), which are explained in section 3.2, to build a better prediction model, in RQ2. We have used Daikon [11] to produce the execution traces from three projects (Commons Lang, Joda Time and JFreeChart). Daikon dynamically detects likely program invariants and it allows to detect properties in C, C++, C#, Eiffel, F#, Java, Perl, and Visual Basic programs [11]. The Java front end (instrumenters) of Daikon, named Chicory, executes the target Java programs and generates the *.dtrace* file that contains the program execution flow along with the variable values in each program point. We have extracted the method sequence calls from the *.dtrace* file generated by Daikon [25].

For the other two projects (Commons Math and Closure Compiler), we have used AspectJ [9] instead of using daikon to produce the trace of method sequence calls, as the test cases from these projects generated very large *.dtrace* files.

Note that although the trace extraction process is different, the format of the extracted method sequence calls is exactly the same. We have collected the method sequences for all modified tests in the current release and also all the failed tests traces from the previous releases [25]. For all the modified test cases in a current release, we have predicted their probability of failing and ranked accordingly.

We have implemented the similarity functions using Java and we have used R [30] to build the regression models and to evaluate the results. We have used "glm" function in R [12] to build the logistic regression model. Further, we have used "predict" function [29] from R to get the response probability value (i.e., probability of a test being passed/failed, in our case) of the observations from the test set.

4.4 Evaluation Metric

To evaluate a list of prioritized test cases, we look at the rank of the first failing test case in the list. An effective prioritization technique is supposed to rank the failing test case higher. Note that this metric is useful when there is only one fault that the test cases are looking for (but there might be still several tests that may detect the fault). This was the case in our study and thus we chose this metric. The more general evaluation metric would be APFD [32], which can be applied for cases with multiple faults.

The defects4j dataset mentions which test case fails in the current version [7]. Therefore, we can easily extract the ranks of first failing test cases provided by different approaches (in this study using regression models and each individual metrics) for each version of a project. Furthermore, we then divided the rank of the first failing test by the total number of modified/new tests in each version, separately. We have done this to calculate the percentage of the test cases that need to be executed in order to catch the first fault in each version.

In case of tied ranks using any measure, we have ranked the tied test cases randomly, by applying the *rank* function in R with a parameter *ties.method="random"* [33]. We have calculated the rank 30 times for each version of the projects to deal with this randomness.

We obtained one normalized rank value between 0 and 1 that represents the percentage of tests required to execute in order to identify the first fault. Since we have calculated the rank 30 times for each version, we have considered the median of these 30 values as the rank of first failing test for each version. We have also combined these median ranks from all the versions of a project and represented them using a boxplot distribution where the lower median line indicates better ranking for the project.

Whenever we compare two distributions of the results and say one outperforms the other, we have first conducted a non-parametric statistical significance test (U-test) [21] to make sure the differences are not due to the randomness of the algorithms. Besides measuring statistical significance, it is also crucial to assess the magnitude of the differences [3]. Effect size measures are used to analyze such a property [3, 14]. In our study, we have used a non-parametric effect size measure, called Vargha and Delaney's $\hat{A}_{12}$ statistics [3, 14, 35]. Given a performance measure M, the $\hat{A}_{12}$ statistics measures the probability that running algorithm X yields higher M values than running another algorithm Y. When the two algorithms are equivalent, then $\hat{A}_{12}$ is 0.5. Therefore, while comparing algorithm X and

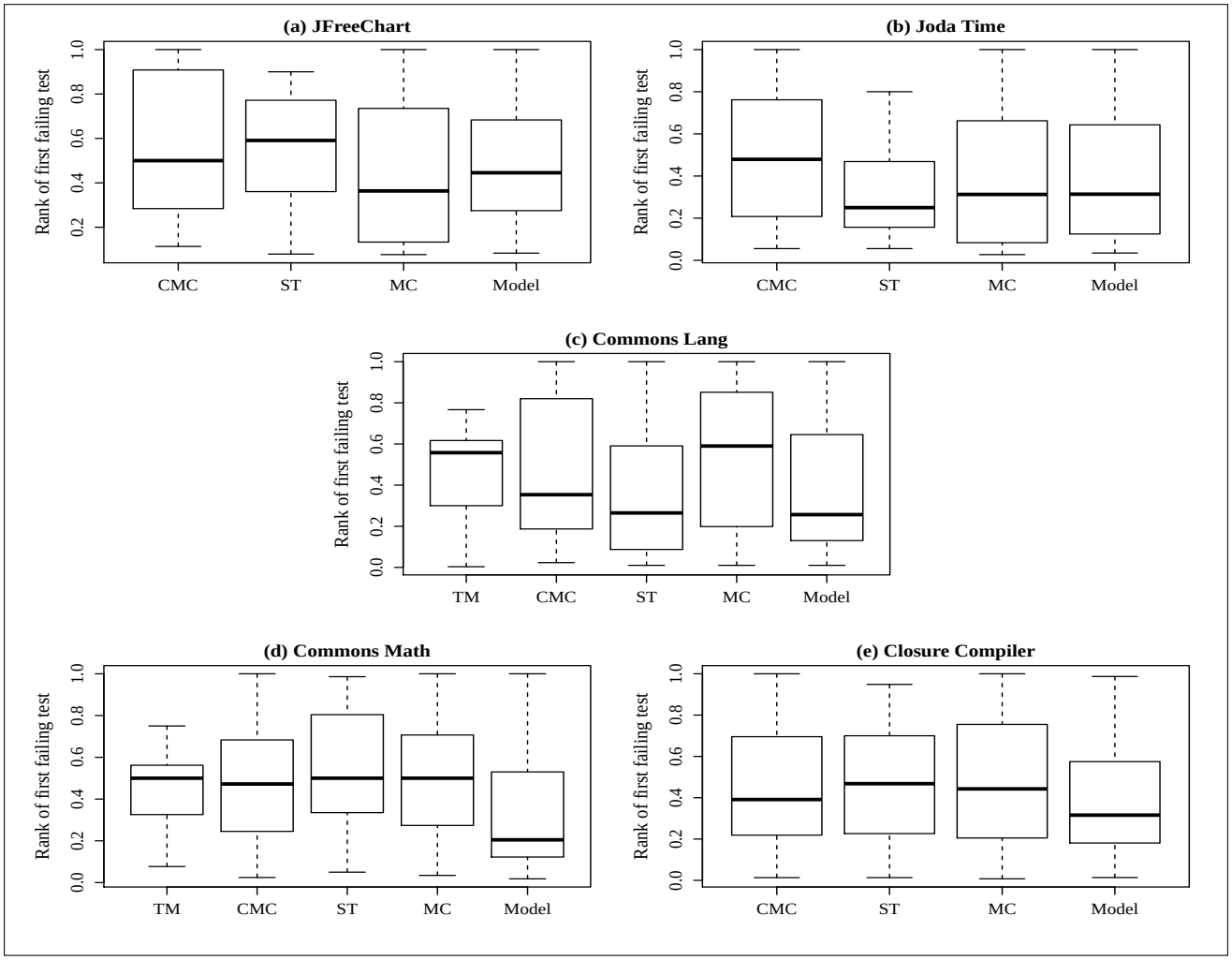

Figure 2: The ranks of the first failing test cases using TM, CMC, MC, ST, and regression model for each version of the project.

Y, $\hat{A}_{12} = 0.8$ represents that we would obtain higher results in 80% of the time with algorithm X compared to the algorithm Y [3, 25].

In our context, the given performance measure M is the percentage of the tests need to be executed (i.e., normalized rank of the first failing test) in order to catch the first fault in a version. Therefore, we get a $\hat{A}_{12}$ value for each version of the project, while comparing two prioritization approaches. Thus, in our case, $\hat{A}_{12} = 0.8$ indicates that approach X ranks the first failing test higher than approach Y in 80% of the time. In other words, approach X can detect the fault faster than approach Y in 80% of time [25]. We have used this $\hat{A}_{12}$ measure to evaluate all of our results.

This is worth mentioning that we have considered only the versions with at least 5 modified/new test cases for the evaluation. We assume that in the versions having less than 5 modified/new test cases, executing all test cases is not costly and hence the prioritization is not actually beneficial [25]. The details of our number of studied versions is mentioned in Table 2.

4.5 Results and Discussion

In the rest of this section, we answer the research questions by comparing both the ranks of the first failing test cases and the $\hat{A}_{12}$ measure. We have also made sure that the differences of the results are statistically significant.

4.5.1 Experimental Results for RQ1. In RQ1, we have compared the ranks of the first failings test cases provided by each individual traditional metric (TM, CM, CMC, ST) and their combined set in a regression model (called Traditional Model). To answer RQ1, we have calculated median rank from 30 different runs for each version and represented median ranks of all versions of a project in a boxplot. Therefore, the lower median line of a boxplot represents better ranking, which indicates the average percentage of tests required to execute in order to catch the first fault of the projects, in each version.

Figure 2 compares the boxplot distribution of the first failing test's rank per project using different approaches. It can be seen that none of the metrics among ST, MC, CMC and TM completely

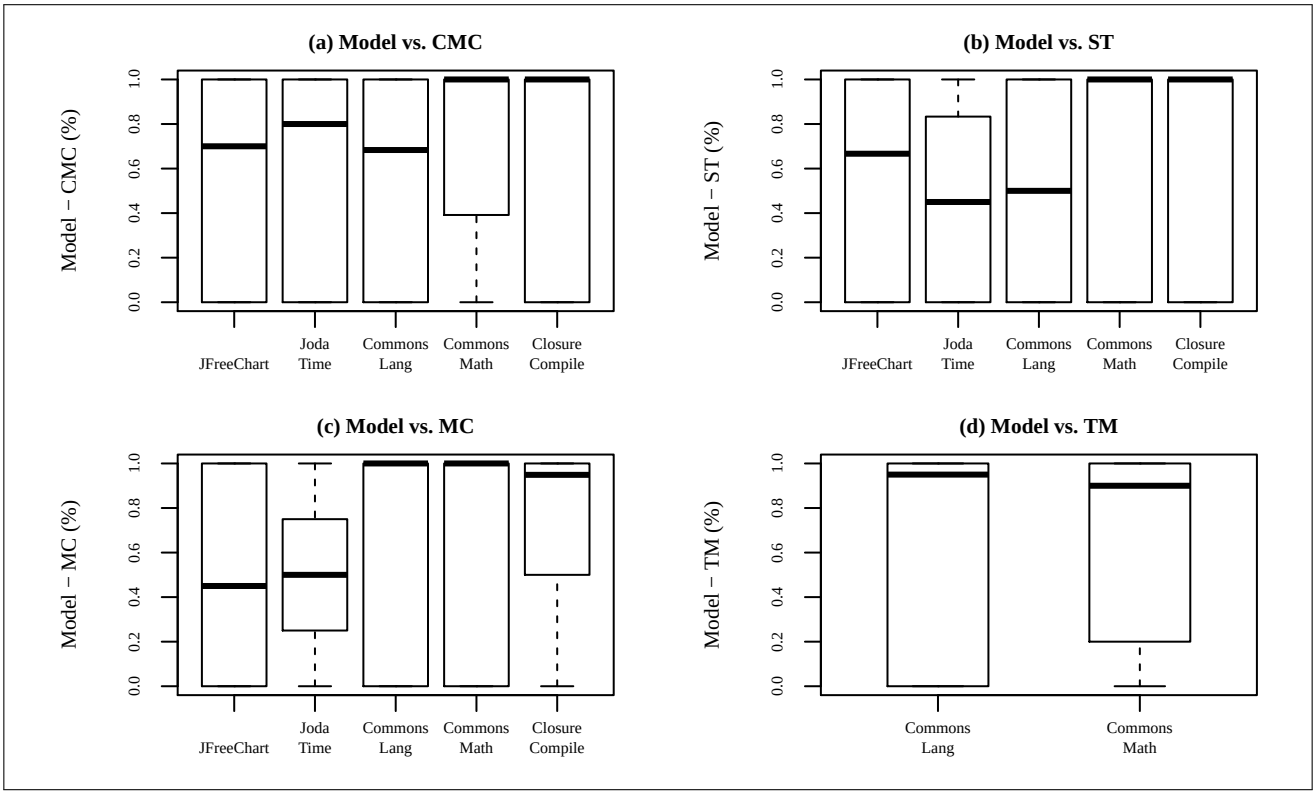

Figure 3: The boxplots of the effect size measures for finding the first fault using CMC, MC, ST, TM and regression model when comparing 30 runs of each versions of each project.

outperforms the others in all the projects. For example, the ranking using ST is better than MC and CMC in Joda Time and Commons Lang project, however, for the other three projects, ST is falling behind the MC and CMC. Similarly, between CMC and MC, the ranking using MC provides better prioritization in JFreeChart and Joda Time project but the opposite behavior is observed for the other three projects except the Commons Math project, where ranking using MC and CMC looks similar.

Looking at the ranks of the first failing tests using the failure probability from the traditional regression model clearly outperforms all the other individual metrics in Commons Math and Closure Compiler project. On average, it requires only 20% and 25% tests to be executed in order to catch the first fault, in Commons Math and Closure Compiler project, respectively . Besides, in Commons Lang project, although the ranking provided by the regression model is better than the others (e.g., MC, CMC), it is equally good as the ST. However, the ranks using the regression model is only falling behind the ST and MC, in Joda Time and JFreeChart project, respectively.

We have also demonstrated the median ranks using the traditional historical fault based metric (TM) for the Commons Lang and Commons Math project. Note that TM does not work in any version of the Closure Compiler and Joda Time project (as mentioned in section 4.3 and Table 2). In addition, in the JFreeChart project, TM works only for 2 versions. Therefore, we have excluded these 3 projects while comparing their performance using TM with the other measures. However, for the other 2 projects, the prioritization using TM also falls behind the other metrics and the prediction model.

Besides showing the median rank of the first failing test, we have also shown the $\hat{A}_{12}$ measure comparing the performance of different prioritization approaches. Figure 3 shows the $\hat{A}_{12}$ measure distribution while comparing the performance of the regression model-based metric with other individual metrics for 30 runs of each version of a project. In the boxplots, the higher median line (i.e., higher than 0.5) represents better prioritization. Therefore, it can be seen from Figure 3(a) that the ranks of first failing test case using the regression model is always better than the CMC in Commons Math and Closure Compiler project, as the median line is in 1.0. The higher than 0.7 median line for the other 3 projects represent that on average the prioritization using the regression model is better than the CMC in 70% of the times.

According to Figure 3(b), prioritization using the regression model is clearly better than the ST in 3 of the projects (e.g., Commons Math, Closure Compiler and JFreeChart). However, the difference is not much significant in the Commons Lang project, as the median line is in 0.5. The only case where the ranking using model slightly falls behind the ST (i.e., the median line is just below the 0.5) is in Joda Time project.

In Figure 3(c), we observe that only for the JFreeChart project, MC performs slightly better than the model (i.e., in 60% of the

cases, as the median line is in 0.4). However, the model clearly outperforms the MC in the other projects, except in Joda Time, where they look similar. In addition, the median line leaning to 1.0 in Figure 3(d) represents that the ranking using the regression model is better than the TM in almost all the cases for Commons Lang and Commons Math project.

Therefore, the results shown in 2 and 3 represents that neither of the traditional individual metrics (e.g., TM, ST, CMC and MC) completely outperforms others for all the projects. However, the prioritization using the combined set of these metrics in a regression model is much more consistent in all the projects.

It can be also seen that in Commons Math and Closure Compiler project, the result using regression model is much better than any other metrics. One plausible reason would be the higher amount of historical data has been used to train the regression model for these 2 projects, which provides better ranks than the use of individual metrics. Note that Commons Math and Closure Compiler projects have data from more versions (106 and 133 versions respectively, mentioned in Table 1) compared to the JFreeChart and Joda Time projects (26 and 27 versions respectively). Therefore, the results suggest that the regression model could provide much better results when it is trained using enough data from longer history.

4.5.2 Experimental Results for RQ2. RQ2 investigates that how well can we predict the first failing test case when combining the similarity based metrics with the traditional metrics. To answer RQ2, we have compared the ranking using predicted ranks from two different logistic regression models. Recall that in the regression model using traditional metrics, we have considered all the traditional metrics (TM, ST, MC and CMC) as the predictors (i.e., independent variables). We call this regression model as the "Traditional Model". In our proposed regression model, we have appended the list of traditional predictors by adding our newly proposed similarity-based metrics (ED, HD, BC and IBC) and therefore, we refer to this regression model as the "Proposed Model".

Figure 4 compares the distribution of the predicted ranks of the first failing tests for all versions of the projects using two models — the traditional and the proposed logistic regression model. Again, we have used the median rank from 30 different runs for each version of the projects. It can be seen that on average, the predicted ranks of the first failing tests from the proposed regression model are always lower than the ranks from the traditional regression model in four projects (JFreeChart, Joda Time, Commons Lang and Commons Math). However, the ranks provided by these two models look similar (the median line is at 0.5) for the Closure Compiler project. The same behavior is also observed in Figure 5 that compares the $\hat{A}_{12}$ measure distribution of the first failing tests ranks from the two regression models.

To summarize, we can say that in our study, combining traditional quality metrics in a regression model improved the prioritization effectiveness. In addition, supplementing the traditional model with similarity-based metrics boosted the effectiveness even more.

4.6 Threats to Validity

In terms of conclusion validity, we have conducted solid experiments to ensure that the results are statistically significant and the magnitude of differences are significant, as well (effect size).

In terms of internal validity, we have used existing libraries and tools as much as possible (e.g., Daikon [11], AspectJ [9]). In terms of construct validity, we use a pretty well-known evaluation metric, which is the rank of first test that catches the fault. The other alternative would be APFD (Average Percentage of Faults Detected) [32], which is commonly used for test prioritization evaluation. However, in our study, each version contains only one fault. So, using APFD would not make sense [25].

In terms of external validity, we have conducted our empirical study based on five real-world open source java libraries from *defects4j* [7] database with several versions and faults [25]. However, generalizing the results to different types of systems may still require further experiments.

5 RELATED WORK

The most related work to this study are those that introduce new test quality metrics. In the rest of this section, we mention some of the related works.

In [36], Xie and Memon mentioned the characteristics for developing a "good" test case/suite that significantly affects the fault-detection effectiveness of the test suite. However, their specific concentration was the test case effectiveness in GUI testing. They found that the tests with more diversity and higher event coverage are better in detecting bugs from the GUI. In another study, Arcuri evaluated the effect of test case sequence length in testing programs with higher internal states [2]. The author showed that the longer test case sequence tend to lead to a higher level of coverage and hence are more effective in detecting faults. We have also considered the test case length (size and executed method sequence length) while combining metrics using the model.

A number of researchers proposed to use historical fault detection as a quality metric, in the context of regression test case selection, prioritization and minimization that makes the regression testing cost-efficient. Kim and Porter [19] proposed to prioritize tests based on historical test execution that also improves the overall regression testing. They considered the number of previous faults exposed by a test as the key prioritizing factor

Park *et al.* considered the test case execution costs and the severity of detected faults to prioritize tests in regression testing [27]. However, instead of specific time frame, the total history of the test execution was used to determine the historical value of the test case. Huang *et al.* also used historical record of test cases execution cost and severity of detected faults [16], however, they applied a search-based approach, i.e., genetic algorithm to generate prioritized test execution order in the current release.

All these traditional historical fault detection measures quality of the test cases and prioritize tests based on their previous fault detection capability retrieved from history. However, we identified the shortcomings of these traditional approaches in our previous work [25] and proposed a better approach that measures quality (and prioritize) of test cases using their similarity to the previously failing tests cases. Our similarity-based metrics also use the historical data; more specifically, it uses failing execution traces from the history. In this study, we have proposed to build the regression model using our proposed similarity-based and other traditional metrics.

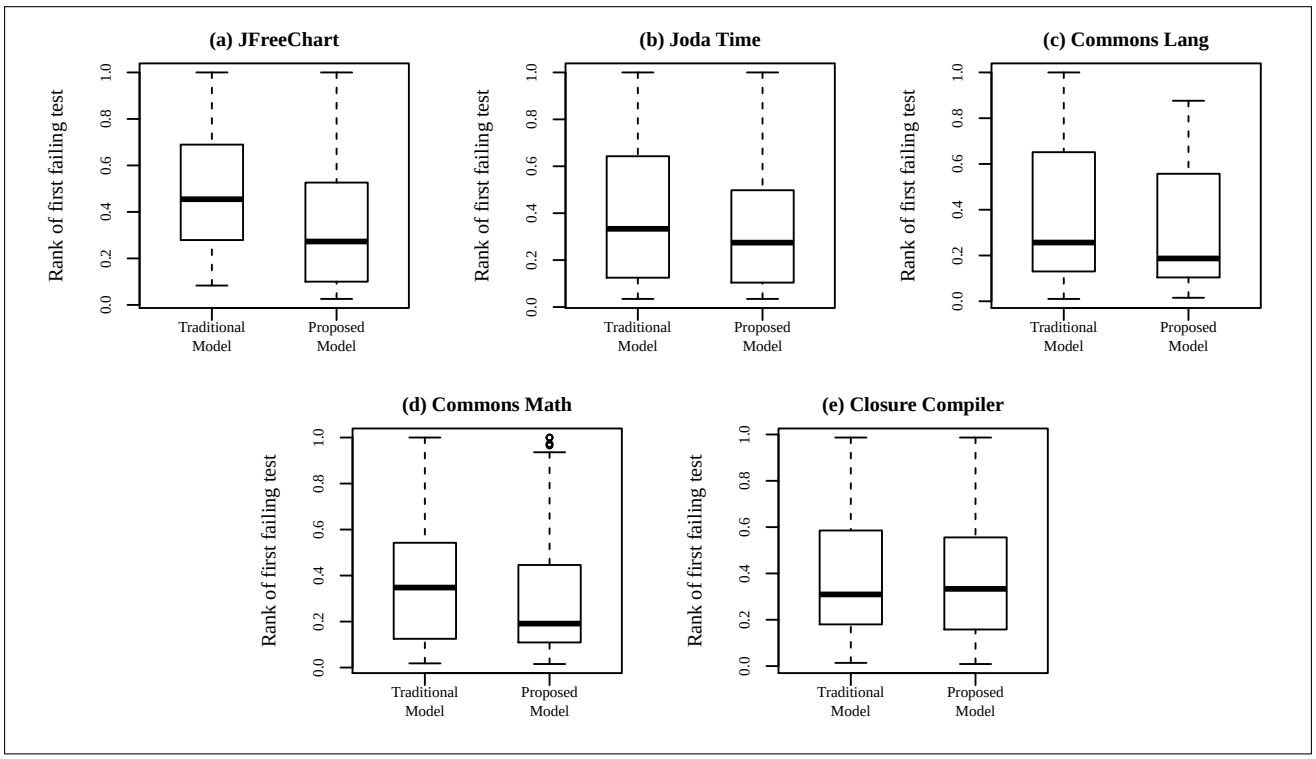

Figure 4: The boxplots of the average rank of the first failing test rank using two prediction models for all versions of each project (Traditional Model: uses all traditional metrics - Proposed Model: uses all traditional and similarity-based metrics).

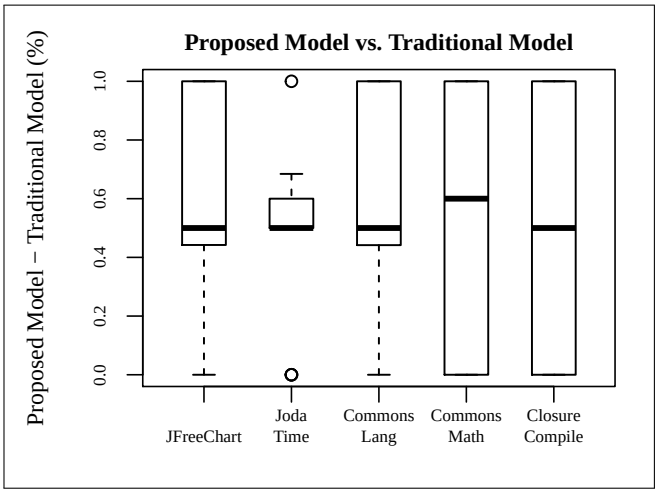

Figure 5: The boxplots of the effect size measures for finding the first fault using the two prediction models, when comparing 30 runs of each versions of each project.

In the fault prediction domain, a number of metrics and approaches have been used in predicting faults in source code. In a large survey [31], Radjenović *et al.* categorized the fault prediction metrics based on size, complexity, OO metrics and process metrics. According to their empirical study, process metrics were found most successful in predicting post-release defects compared to the traditional source code metrics (e.g., LOC and complexity metrics) and OO metrics (e.g., CK metrics [5]). Process metrics are usually extracted from the combination of source code and repository. Some of the process metrics are number of revisions, bug fixes, refactoring, code, delta, code churn (i.e., sum of added and deleted lines of code over all revisions) etc.

Nagappan *et al.* proposed a set of 9 test quantification and complexity metrics and Object-Orientation (OO) metrics to evaluate Junit test cases in terms of early estimation of software defects [23]. Test quantification metrics evaluate the test cases by the amount of the tests (e.g., number of assertions or LOC) written to check the program thoroughly. We have also used the size of tests (i.e., LOC) and coverage (i.e., method coverage) for our comparison.

In another work [22], Nagappan *et al.* combined the complexity metrics using a regression model in order to predict the post-release defects. They found that there is no single metric that works good in different projects, in terms of detecting post release defects. Therefore, they combined the metrics using a prediction model to predict post-release defects. They also mentioned about the *multicollinearity* problem while combining several metrics and they applied principal component analysis (PCA) to overcome the problem. In our study, we have also dealt with the *multicollinearity* problem, however, we have applied different approach (considered Variance Inflation Factor (VIF)) of the model is below 2.5) as suggested by Shihab *et al.* [4, 34]. We have also considered another findings from [22], which is predictors should be obtained from the

same or similar projects. Therefore, we have used different quality measures of a project to build the regression model for the same project. Similar to [22], we have also proposed to combine several metrics using a regression model, however, we have used the model to predict the failing probability of test cases and prioritized the tests based on higher failing probability value.

6 CONCLUSION

Test case prioritization ranks the tests based on some quality measures to detect the potential faults earlier. In this paper, we have studied the prioritization effectiveness using some individual quality metrics. We have also prioritized the test cases based on their failure probability using a regression model that combines the individual metrics. We have conducted a large empirical study (247 versions from five real-world java projects with real faults) and compared the ranks of fault revealing tests using different approach in the context of test case prioritization. We have not found any individual metric that is always good in prioritizing the fault reveling tests, however, the combined set of the metrics in a regression model preformed better in all the projects. We have also proposed to add our recently introduced similarity-based metrics while building the regression model, as the results show adding the similarity-based metrics provides better prioritization.

In the future, we will extend our study by adding more quality measures to prioritize test cases. Moreover, we are also interested to apply our proposed combined set of metrics as criteria to build an automatic test generation tool that can generate high-quality tests.

REFERENCES

[1] Jeff Anderson, Saeed Salem, and Hyunsook Do. 2014. Improving the effectiveness of test suite through mining historical data. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 142–151.

[2] Andrea Arcuri. 2010. Longer is better: On the role of test sequence length in software testing. In *Third International Conference on Software Testing, Verification and Validation (ICST), 2010*. IEEE, 469–478.

[3] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *33rd International Conference on Software Engineering (ICSE), 2011*. IEEE, 1–10.

[4] Marcelo Cataldo, Audris Mockus, Jeffrey Roberts, James D Herbsleb, et al. 2009. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering* 35, 6 (2009), 864–878.

[5] Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 20, 6 (1994), 476–493.

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

[7] defects4j. http://homes.cs.washington.edu/~rjust/defects4j/. (????). [Online; last accessed 26-Dec-2015].

[8] G. Dong and J. Pei. 2007. *Sequence Data Mining*. Springer US. https://books.google.ca/books?id=GESJmZpkePIC

[9] Eclipse-AspectJ. The AspectJ Project. https://eclipse.org/aspectj/. (????). [Online; last accessed 26-Dec-2015].

[10] Eclipse-JDT. Eclipse java Development Tool (JDT). https://eclipse.org/jdt//. (????). [Online; last accessed 26-Dec-2015].

[11] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1 (2007), 35–45.

[12] GLM. https://cran.r-project.org/web/packages/glm2/glm2.pdf. (????). [Online; last accessed 26-Dec-2015].

[13] John B. Goodenough and Susan L. Gerhart. 1975. Toward a theory of test data selection. *IEEE Transactions on Software Engineering* 2 (1975), 156–173.

[14] Keith J Goulden. 2006. Effect Sizes for Research: A Broad Practical Approach. (2006).

[15] D. Gusfield. 1997. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press. https://books.google.ca/books?id=Ofw5w1yuD8kC

[16] Yu-Chi Huang, Kuan-Li Peng, and Chin-Yu Huang. 2012. A history-based cost-cognizant test case prioritization technique in regression testing. *Journal of Systems and Software* 85, 3 (2012), 626–637.

[17] Yue Jia and Mark Harman. 2011. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678.

[18] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), 2014*. ACM, 437–440.

[19] Jung-Min Kim and Adam Porter. 2002. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24rd International Conference on Software Engineering (ICSE), 2002*. IEEE, 119–129.

[20] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. 2007. Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering (ICSE), 2007*. IEEE Computer Society, 489–498.

[21] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[22] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. 2006. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*. ACM, 452–461.

[23] Nachiappan Nagappan, Laurie Williams, Mladen Vouk, and Jason Osborne. 2007. Using in-process testing metrics to estimate post-release field quality. In *18th International Symposium on Software Reliability (ISSRE), 2007*. IEEE, 209–214.

[24] Tanzeem Noor and Hadi Hemmati. 2015. Test case analytics: Mining test case traces to improve risk-driven testing. In *IEEE 1st International Workshop on Software Analytics (SWAN), 2015*. IEEE, 13–16.

[25] Tanzeem Bin Noor and Hadi Hemmati. 2015. A similarity-based approach for test case prioritization using historical failure data. In *26th International Symposium on Software Reliability Engineering (ISSRE), 2015*. IEEE, 58–68.

[26] M.A. P. 2008. *Foundations of Software Testing*. Pearson Education. https://books.google.ca/books?id=yU-rTcurys8C

[27] Hyuncheol Park, Hoyeon Ryu, and Jongmoon Baik. 2008. Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing. In *Second International Conference on Secure System Integration and Reliability Improvement (SSIRI), 2008*. IEEE, 39–46.

[28] M. Pezzè and M. Young. 2008. *Software testing and analysis: process, principles, and techniques*. Wiley.

[29] Predict. https://stat.ethz.ch/R-manual/R-devel/library/stats/html/predict.glm.html. (????). [Online; last accessed 26-Dec-2015].

[30] R-Project. The R Project for Statistical Computing. http://www.r-project.org/. (????). [Online; last accessed 26-Dec-2015].

[31] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. 2013. Software fault prediction metrics: A systematic literature review. *Information and Software Technology* 55, 8 (2013), 1397–1418.

[32] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test case prioritization: An empirical study. In *Proceedings of IEEE International Conference on Software Maintenance, 1999.(ICSM'99)*. IEEE, 179–188.

[33] R:Ranks. R:Sample Ranks. https://stat.ethz.ch/R-manual/R-devel/library/base/html/rank.html. (????). [Online; last accessed 26-Dec-2015].

[34] Emad Shihab, Audris Mockus, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2011. High-impact defects: a study of breakage and surprise defects. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 300–310.

[35] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[36] Qing Xie and Atif M Memon. 2006. Studying the characteristics of a "Good" GUI test suite. In *17th International Symposium on Software Reliability Engineering (ISSRE), 2006*. IEEE, 159–168.

[37] Shin Yoo, Robert Nilsson, and Mark Harman. 2011. Faster fault finding at Google using multi objective regression test optimisation. In *8th European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'11), Szeged, Hungary*.

[38] Hong Zhu, Patrick AV Hall, and John HR May. 1997. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)* 29, 4 (1997), 366–427.

[39] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting defects for eclipse. In *International Workshop on Predictor Models in Software Engineering, PROMISE'07: ICSE Workshops 2007*. IEEE, 9–9.