A Similarity-based Test Case Quality Metric using Historical Failure Data

by

Tanzeem Bin Noor

A thesis submitted to
The Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

MASTER OF SCIENCE

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
December 2015

Thesis advisor

Dr. Hadi Hemmati

Author

Tanzeem Bin Noor

A Similarity-based Test Case Quality Metric using Historical Failure Data

Abstract

A test case is a set of input data and expected output, designed to verify whether the system under test satisfies all requirements and works correctly. An effective test case reveals a fault when the actual output differs from the expected output (i.e., the test case fails). The effectiveness of test cases is estimated using quality metrics, such as code coverage, size, and historical fault detection. Prior studies have shown that previously failing test cases are highly likely to fail again in the next releases; therefore, they are ranked higher. However, in practice, a failing test case may not be exactly the same as a previously failed test case, but quite similar. In this thesis, I have defined a metric that estimates test case quality using its similarity to the previously failing test cases. Moreover, I have evaluated the effectiveness of the proposed test quality metric through detailed empirical study.

Acknowledgments

I am grateful to the almighty God for giving me the strength to pursue my graduate studies staying away from my family for more than two years.

I would like to thank my supervisor, Dr. Hadi Hemmati, for his continuous support and careful guidance for the entire duration of my Master's program at the UofM. I am grateful to the department of Computer Science, UofM for providing me the financial support through Guaranteed Funding Package (GFP). Further, I would also like to acknowledge the additional financial support of the University of Manitoba Graduate Fellowship (UMGF) and again thanks to Dr. Hemmati for the assistance provided.

I would like to thank my committee members, Dr. Yang Wang and Dr. Ian Jeffrey for their careful review and valuable feedback. I would also like to thank Md Mahfuzur Rahman for his help and advice.

Last but not the least, I would like to acknowledge the love and support of my parents.

Publications

Some ideas and figures in this thesis have previously appeared in the following publications by the author:

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

2. Tanzeem Bin Noor and Hadi Hemmati. A similarity-based approach for test case prioritization using historical failure data. In 26th IEEE International Symposium on Software Reliability Engineering (ISSRE), 10 pages, 2015.

This thesis is dedicated to my parents.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software testing is a crucial task in the software development process that ensures software quality. The goal of testing is to verify the software under development against the client's requirements and identify its faults. Generally, numerous test cases can be designed and executed to verify the software. Among these huge sets of test cases, only a few effective tests can detect faults (mismatches between actual and expected output). We call a test case effective if it can detect a fault. Since it is impossible to verify a real-world software with all possible test cases, we need to identify effective test cases.

The ultimate effectiveness measure of a test case is its actual fault-detection power that indicates how many real faults the test case can detect. 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).

Test case prioritization is an important element in software quality assurance in practice. It is even more crucial in modern software development processes, where

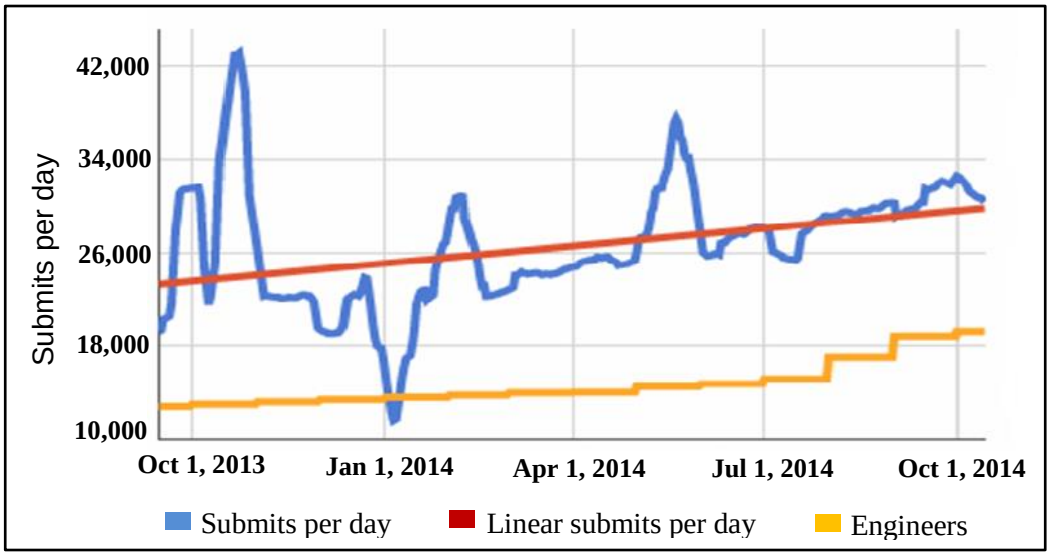

Figure 1.1: Number of tests submitted at Google over time [Mehta]

continuous integration is widely followed. Continuous integration suggests the developers to frequently merge their codes with the main code repository. Codes are frequently changed due to either adding new functionality or to fix the bugs. A study shows that in Google, more than 20 code changes occur in every minute [Yoo et al., 2011]. 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. Figure 1.1 shows the growth of tests submitted at Google over time [Mehta].

On the other hand, 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 tests higher so that the faults can be detected by executing only the higher ranked tests. 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.

Code coverage is a widely used quality metric that measures how much of the code (e.g., number of lines, blocks, conditions etc.) from the program is exercised during the tests execution. As faulty code needs to be executed to reveal its fault, covering (executing) more code increases the probability of covering the faulty code, as well. However, covering the faulty code may not always result in detecting its faults [Pezzè and Young, 2008]. Faults are only revealed when the faulty code is executed with special input values, which actually causes the tests to fail. Therefore, code coverage does not guarantee detecting faults and is simply a heuristic that estimates the test case quality.

Detecting previous faults is another important factor used to estimate the tests case quality. In the context of regression testing, test cases can be prioritized based on the previous faults (history-based test prioritization) [Ammann and Offutt, 2008]. The rationale behind it is that if a test detects a fault in the past, it is probably touching a part of the code that used to be faulty. On the other hand, defect prediction studies have shown that if a file/method used to be faulty, it is highly likely to be faulty again, specially if it is being changed [D'Ambros et al., 2010; Zimmermann et al., 2007]. So the test case that touches those faulty places might detect new faults,

as well. The type of quality metric that quantifies this concept is called history-based quality metric, in this thesis.

The typical history-based quality metric basically goes through the history of the software and identifies test cases from the current release that used to fail in any of the previous releases [D'Ambros et al., 2010; Zimmermann et al., 2007]. Those previously failed test cases are ranked higher and considered "important", in a sense that it may also reveal a fault in the current release. The problem with this approach is that in many situations (e.g., when a new test is added or when an old test is slightly modified to catch an undetected fault) the test case of the current release that detects a fault is not exactly the same as any of the previously failing test cases. However, it is quite similar to one (or more) old failing test case(s), in terms of the sequence of methods being called (these similar test cases are verifying different aspects of a risky scenario with minor differences).

In this thesis, I have defined a set of test case quality metrics that assign historical faultiness values to the test cases, when they are similar to the failing tests from previous releases. Each metric defines similarity using a different similarity/distance function, but they all are applied on the test cases' execution traces. I look at an execution trace as a sequence of method calls from the program source code, when the test case is executed. I have conducted a series of empirical studies (using five open source java software systems) to compare the effectiveness of these metrics in the context of identifying the first fault (i.e., first failing test case) of a system.

I have found that the similarity-based quality metrics are more effective in identifying the first failing test cases compared to the traditional history-based quality

metric. Moreover, the results also show that the similarity-based quality metrics are also better than other test quality metrics, e.g., coverage, test size and change related metrics in prioritizing fault-revealing tests. I have also found that combining the similarity-based metrics with the existing metrics in a prediction model may result in a better test quality metric when enough historical data are available.

The rest of this thesis is organized as follows. Chapter 2 mentions background and some existing traditional test quality metrics. I have explored some related works in chapter 3. The motivation and problem description of this thesis, and my proposed similarity-based test case quality metrics have been explained in chapter 4. In chapter 5, I have described detailed empirical study and its results. Finally, I have concluded in chapter 6 by summarizing the contribution of my thesis and future works.

Chapter 2

Background

The primary goal of using test case quality metrics is to evaluate the tests and find the scope of improvement required in the test cases. In the applications like test case prioritization and generation, different test case quality metrics are extensively used. The primary quality measure of a test case is its ability to detect software faults, i.e., whether the test fails on the program. Sometimes the severity of the revealed faults might be a crucial factor and therefore the tests that detect more severe faults be considered as higher quality.

As the ultimate goal of testing is detecting faults, any measure that directly quantifies the fault detection power is a perfect metric, in terms of effectiveness [Rothermel et al., 1999]. However, the actual fault detection metric can be calculated only if we know beforehand how many faults are in the source code. Then we execute the tests on the code and check how many of those faults are detected by each test. Unfortunately, this is not practical, since firstly, we do not know the number of faults beforehand and secondly, we often need to measure the quality of the test cases be-

fore executing them (e.g., in the context of test prioritization). Therefore, several heuristics are used to define test quality metrics with the hope that they have high correlations with real fault detection power of the test cases. A well-known set of such metrics are test adequacy criteria.

Test adequacy criteria are usually used for evaluating test cases/suites. It is also considered as a guiding mechanism in automatic test generation tools for generating high quality tests. 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 criterion [Goodenough and Gerhart, 1975]. Two main types of test adequacy criteria are coverage-based test adequacy criteria and fault-based test adequacy criteria [P, 2008; Pezzè and Young, 2008]. Besides the test adequacy criteria, other measures having high correlation to the fault detection are also considered as test quality metrics such as previous fault detection, size, complexity, code churn, etc. [Zimmermann et al., 2007]. In the rest of this chapter, I explain the test adequacy criteria and the other quality metrics.

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 reveals a fault when it executes a segment from the program that causes a failure. Most existing automated test generation tools [Fraser and Arcuri, 2014; Sen and Agha, 2006] try to generate test cases that cover/execute 100% (or as close as possible to that) of the source code. Therefore, high coverage has always been an indicator of good quality because

executing (covering) more segments by a test case also increase the probability of detecting faults. Coverage based test adequacy criteria is further grouped into control-flow coverage and data-flow coverage.

2.1.1 Control-flow Coverage

Coverage-based tools use control flow analysis to measure the adequacy of test cases. The basic idea is to examine the execution of test cases in terms of their coverage of all possible execution paths in the flow-graph model of the program [Zhu et al., 1997]. Adequacy criteria based on control-flow could be categorized into number of coverage levels. Some of the major categories are as follows:

Procedure/Method Coverage:

The key question here is "has each function (or method) in the program been called?". For each test case, the method coverage refers to the number of methods called from the test case divided by the total number of methods in the program [P, 2008].

Statement Coverage:

A finer-grained criterion compared to method coverage is the statement coverage, which is the number of executed statements, by the test cases, divided by the total number of statements in the program/class under test. Statement coverage is still a weak coverage criterion [Zhu et al., 1997] and yet the most used one in practice [P, 2008; Pezzè and Young, 2008].

Branch Coverage:

The branch coverage examines all branches (both the true and false cases) of the program. The branch coverage for each test is calculated by dividing the number of executed branches by the total number of branches. Branch coverage is stronger criteria than the statement coverage and it actually subsumes the statement coverage [Zhu et al., 1997].

Path Coverage:

Execution of all branches does not mean that all the combinations of control transfers have been checked. The requirement of checking all combinations of branches is usually called path coverage [Zhu et al., 1997]. The path coverage of a test case is the fraction of paths of the program executed by the test [Pezzè and Young, 2008].

Condition Coverage:

The basic condition coverage of a test case is the fraction of total number of truth values assumed by the basic conditions of program during execution of the test case [Pezzè and Young, 2008]. Condition coverage can be extended to multiple condition coverage that includes both the basic condition and branch adequacy criteria. It requires checking each possible evaluation of the compound conditions, i.e., considers each sub-conditions [Zhu et al., 1997].

2.1.2 Data-Flow Code Coverage

Another coverage-based test adequacy criterion is data-flow coverage, which depends on the flow of data through the program. Data-flow coverage criterion focuses on the values associated with variables and how these associations can affect the execution of the program [Zhu et al., 1997]. This analysis emphasizes the occurrences of variables within the program and each variable occurrence is classified as either a definition occurrence *(def)* or a use occurrence *(use)*. A definition occurrence of a variable is where a value is assigned to the variable. A use occurrence of a variable is where the value of the variable is referred. There are several types of data flow coverage criteria. 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 [Pezzè and Young, 2008].

2.2 Fault-based Test Adequacy Criteria

Fault-based adequacy criteria take another approach than code coverage and measure the quality of the tests by their ability to detect known faults, as an estimate for their ability for detecting unknown faults [Jia and Harman, 2011]. The known faults are either real faults or artificial faults (mutants). The real faults are the actual faults which are intentionally seeded into the code, for the sake of evaluation. On the other hand, a mutant is a variant of the original program, automatically created by making a small syntactic change (e.g., a condition, a statement, a variable) in the original program [Pezzè and Young, 2008].

In mutation testing, tests are executed on all of the mutant programs and a mutant is killed if it fails on the test case. 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 is also more effective in finding real faults in the program.

2.3 Other Quality Metrics

Besides the test adequacy criteria (code coverage and fault-based measures), there are some other quality metrics that have high correlation to fault detection ability of the test cases, such as historical fault detection, size of the test, complexity, code churn, etc. [D'Ambros et al., 2010]. Most of these metrics are originated from the defect prediction domain and/or regression testing.

Regression testing is the process of re-testing the software that has been modified due to either fault fixes or adding new features [Ammann and Offutt, 2008]. Regression testing ensures that anything that was working before is still working and is not regressed due to the modification, unless the regression is indeed the goal of the modification. For instance, during unit testing, every time a unit, such as a class, is modified by adding new methods, regression testing should ensure that the unmodified methods also work as required. As regression testing requires re-testing the existing elements, test case selection, minimization and prioritization are very crucial factors, for saving the testing resources such as the number of testers required, time, budget etc.

Commonly used quality metrics to evaluate tests in the regression testing are code coverage (i.e., coverage-based test adequacy criteria as described in section 2.1),

historical fault detection and code coverage of the changed parts of the program. Some of these other quality metrics are defined below.

2.3.1 Historical Fault Detection

Historical fault detection metrics assign higher priority to test cases of the current release that failed historically, i.e., on any previous release [D'Ambros et al., 2010; Zimmermann et al., 2007; Kim et al., 2007]. The reason 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. Therefore, in my thesis, I consider this historical fault based metric as traditional metric that counts previous failing occurrences of a test case.

2.3.2 Code Coverage of the Changed Parts

Since in regression testing the source code is modified from the previous version, metrics that measure the coverage of the changed parts of the code are very important [D'Ambros et al., 2010], because even if the total coverage of the test suite is not high, we still expect a high coverage in the changed part to assure proper regression testing. This metric prioritizes the test cases that execute any change part of the code directly or indirectly.

2.3.3 Size of Tests

Finally, the size of the test cases is also a commonly used test quality metric, where the large size indicates a more effective test. Generally, size of a test case refers to the LOC (Line of Codes) in the test method. However, sometimes the number of

assertions in a test case is considered a better size measure, since it directly measures the amount of verifications applied by the test cases. This is to mention that size of test cases is not the same as its coverage [Pezzè and Young, 2008]. For example, test t_1 can cover a method with one assertion and test t_2 do the same but try it with 5 more assertions. In this case, test t_2 has higher chances to detect faults than t_1.

Chapter 3

Related Work

Test case/suite quality has been studied in different domains such as test case prioritization, test case generation, fault prediction etc. A number of quality metrics have been proposed so far in order to evaluate test case quality from these perspectives. In the rest of this chapter, I mention some of the related works in three categories.

3.1 Test Case Effectiveness Metrics

In [Xie and Memon, 2006], the authors mentioned the characteristics for developing a "good" test case/suite that significantly affect the fault-detection effectiveness of a test suite. Their specific concentration is the test case effectiveness in GUI testing, which considers test cases as sequences of GUI input events that a user might perform on a GUI. They found that the tests with more diversity and higher event coverage are better in detecting faults from the GUI.

In another study, Arcuri evaluated the effect of test case sequence length in testing programs with higher internal states [Arcuri, 2010]. 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.

Nagappan et al. [2007] proposed a set of 9 metrics to evaluate Junit tests in terms of early estimation of software defects. The metrics were divided into two groups: test quantification metrics, and complexity and Object-Orientation (OO) metrics. Test quantification metrics evaluated the tests by the amount of the tests (e.g., number of assertions or LOC) written in order to check the program source code thoroughly. Besides, the complexity and OO metrics such as Weighted methods per class (WMC), Number of children (NOC), Depth of inheritance tree of a class (DIT) etc. were chosen from a widely used Chidamber-Kemerer (CK) OO metric suite [Chidamber and Kemerer, 1994]. Nagappan *et al.* examined the relative ratio of the tests to the source code with the complexity and OO metrics.

In this thesis, I have examined some of these common metrics such as coverage and test length.

3.2 Fault Prediction Metrics

Fault prediction is a widely studied area in software testing research. A number of metrics and approaches are used in predicting faults in source code. In a large survey, Radjenović et al. [2013] categorized the bug prediction metrics based on size, complexity, OO metrics and process metrics. 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. According to the survey, 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 [Chidamber and Kemerer, 1994]).

Nagappan and Ball also found the code churn metrics successful in fault prediction [Nagappan and Ball, 2005]. However, they showed that the relative code churn metric was much better than the absolute code churn metric. In relative code churn metric, churn measures are normalized values of the various measures obtained during the churn process.

Shihab *et al.* studied the impact of change-metrics in terms of risk management of a software [Shihab et al., 2012]. A change is considered risky when they might result in some faults in the future. The authors found that the number of lines and chunks of code added by the change, the developer experience, the number of bug reports linked to a change and the faultiness of the files being changed are the best indicators of change risk.

Moser also mentioned the number of bug fixes as one of the powerful fault-predictor process metrics [Moser et al., 2008]. He showed that the previous bug-fixing activities are likely to introduce new faults in the later releases. Similar to this, Zimmermann et al. [2007] also showed that the number of past bug fixes extracted from the repository is also correlated with the number of future fixes.

Most of the fault prediction studies are applied on the source code files. However, I have used two main findings from these studies. 1) The previously faulty source

code are likely to have faults again in the new release. This is the main motivation behind the historical fault detection metrics that I have explored in this thesis. 2) Changing a source code increases its chance to fail. Therefore, the test case that covers (executes) more changed part of the source code is better. So, in this study, I have considered the number of changed methods covered by a test as one of the existing test case quality metrics and I call this metric "Changed Method Coverage".

3.3 Historical Data Used in Metrics

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 in terms of budget, time and number of testers. Kim and Porter [2002] 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

Some history-based metrics used a partial and more recent history based on time-window whereas other metrics used the full historical data. For example, in the context of regression testing under continuous integration, Elbaum *et al.* proposed a regression testing technique that use time windows to track how recently test suites have been executed and revealed failures, to select and prioritize tests [Elbaum et al., 2014]. They used the readily available test execution data from the recent history data to determine higher priority tests for execution.

On the other hand, Park *et al.* considered the test case execution costs and the severity of detected faults to prioritize tests in regression testing [Park et al., 2008].

However, instead of specific time frame, the total history of the test execution was used to determine the historical value of the test case. The assumption was that the costs of the test cases execution and the fault severity of detected faults can significantly change from one release and therefore, the complete history can further improve test case prioritization. Huang *et al.* also used historical record of test cases execution cost and severity of detected faults [Huang et al., 2012], however, they applied a search-based approach, i.e., genetic algorithm to generate prioritized test execution order in the current release.

Khalilian et al. [2012] proposed a prioritizing equation with variable coefficients gained according to the available historical test case data performance. Their history-based technique improved the overall test prioritization compared to the approach used by Kim and Porter.

All these traditional historical fault detection measures prioritize the test cases that detect faults in previous releases. However, these traditional approaches have some shortcomings. In the traditional historical fault-based quality metrics, a test case from the current release would be considered as effective if the exact same test also failed in the previous release [D'Ambros et al., 2010; Zimmermann et al., 2007].

In this thesis, I argue (and empirically show) that there might be several failing test cases, in the current release, that are not exactly the same as those old fault revealing test cases, but they are quite similar to them. Therefore, I have addressed this issue by proposing a test quality metric that would prioritize the test cases that are similar to the previous fault revealing tests, while considering the history of the software.

Chapter 4

Methodology

In this chapter, at first, I describe the problem that is targeted in this thesis by using a motivational example, where the traditional history-based metric fails to rank the tests properly. Then I give an overview of my proposed similarity-based quality metric that ranks the tests of a current version based on their similarity to the previously failing tests.

4.1 Motivation and Problem Description

Test case quality metrics are used in different applications; most commonly in evaluating existing test suites that try to make sure enough testing has been done. An automatic test case generation tool also uses quality metrics to evaluate test case effectiveness in order to produce high quality tests. In addition, quality metrics are used in prioritizing test cases, when the resource (e.g., time, number of software testers) is limited. A test case prioritization technique ranks the test cases based

on the quality metrics so that the more effective (higher quality) tests are being executed first and detect the software faults faster, within the limited testing budget. Test case prioritization is very important in practice for software companies, specially when continuous integration and rapid release demands fast development paces.

In continuous integration development environments, new or changed code are frequently integrated with the mainline codebase. Continuous integration processes require extensive testing to be performed prior to code submission. To make this process cost-effective, regression testing techniques must operate effectively within continuous integration development [Elbaum et al., 2014]. Recently, Elbaum *et al.* has shown that in the continuous integration process, test selection and prioritization techniques can be performed, cost-effectively, in the absence of coverage data by using readily available test suite execution history [Elbaum et al., 2014]. Test execution history provides the information regarding a test, i.e., whether it passed or failed (detected a fault, previously). In addition, the historical fault detection measure is also a guiding factor to detect faults in the current version [D'Ambros et al., 2010; Zimmermann et al., 2007; Kim et al., 2007; Anderson et al., 2014].

In the traditional history-based quality metrics, a test case from the current release would be considered as effective if the exact same test also failed in the previous releases [D'Ambros et al., 2010; Zimmermann et al., 2007]. Now assume a test case, such as testLang747 (a test case from the latest version of project Commons Lang, a software system used for my study, explained in section 5.2) in Fig. 4.1, that is just added to the current test suite and actually fails (detects a fault). This test case is not effective according to the traditional history-based quality metric, since it did

```
@Test
public void testLang747() {
       assertEquals(Integer.valueOf(0x8000), NumberUtils.createNumber("0x8000"));
       assertEquals(new BigInteger("8000000000000000", 16),
       NumberUtils.createNumber("0x8000000000000000"));
       ....
       assertEquals(new BigInteger("FFFFFFFFFFFFFFFF", 16),
       NumberUtils.createNumber("0xFFFFFFFFFFFFFFFF"));
       assertEquals(Long.valueOf(0x80000000000000L),
       NumberUtils.createNumber("0x00080000000000000"));
       assertEquals(Long.valueOf(0x800000000000000L),
       NumberUtils.createNumber("0x08000000000000000"));
       ...
       assertEquals(Long.valueOf(0x7FFFFFFFFFFFFFFL),
       NumberUtils.createNumber("0x07FFFFFFFFFFFFFF"));
       assertEquals(new BigInteger("8000000000000000", 16),
       NumberUtils.createNumber("0x0008000000000000000"));
       assertEquals(new BigInteger("FFFFFFFFFFFFFFFF", 16),
       NumberUtils.createNumber("0x0FFFFFFFFFFFFFFF"));
}
```

Figure 4.1: A failing test case in the Commons Lang project's latest version

```
@Test
public void testStringCreateNumberEnsureNoPrecisionLoss() {
       String shouldBeFloat = "1.23";
       String shouldBeDouble = "3.40282354e+38";
       String shouldBeBigDecimal = "1.797693134862315759e+308";
       ... .. . .
       assertTrue(NumberUtils.createNumber(shouldBeFloat) instanceof Float);
       ..... . ..
       assertTrue(NumberUtils.createNumber(shouldBeDouble) instanceof Double);
       assertTrue(NumberUtils.createNumber(shouldBeBigDecimal) instanceof BigDecimal);
}
```

Figure 4.2: A failing test case in the previous versions of the Commons Lang project that is similar to the test case in Fig. 4.1

not exist in the previous releases, to fail. However, there are some test cases in the past that are quite similar to this test case and they failed, e.g., the test case in Fig. 4.2. Therefore, it would be nice to have a history-based quality measure for test cases that do not only look at exact occurrences of the test case in the past, but look at its similar cases, as well. The key question here is "how do we identify such similar test cases?".

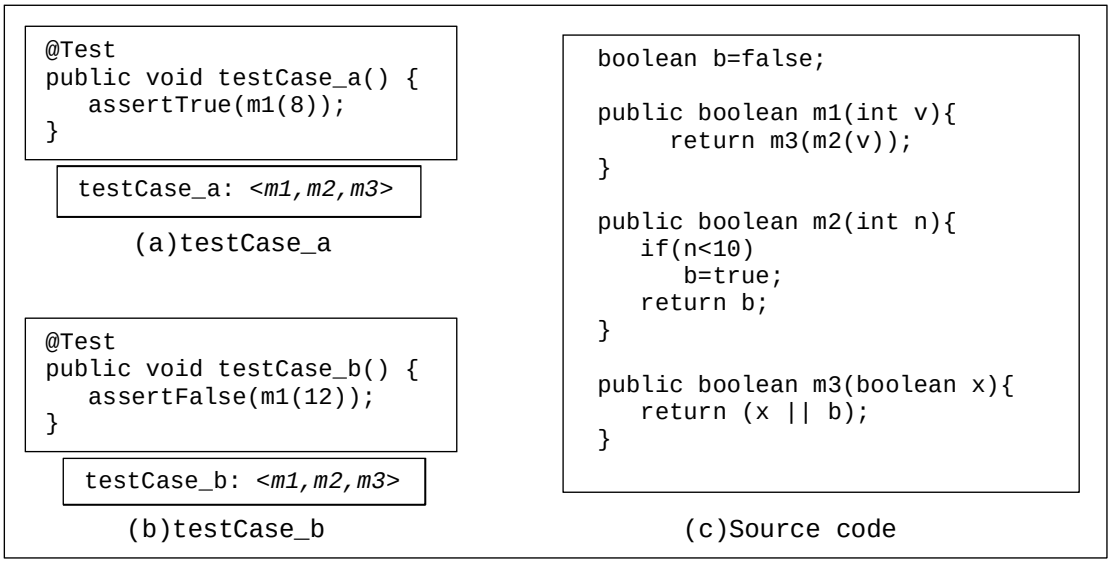

Figure 4.3: Example of test cases and their corresponding method calls

Usually, along with the historical fault detection information, execution traces can also be collected from the history. In general, an execution trace of a test case is the sequence of method calls from the program source. For example, when a Junit test "testCase_a" in Fig. 4.3a is executed, it calls method $m1$ from the program source code in Fig. 4.3c and its execution trace would be $testCase\_a < m1, m2, m3 >$ in terms of method invocation sequence. Now, assume another Junit test "testCase_b" in Fig. 4.3b. This test seems different from the test in Fig. 4.3a, since both its name and assert statement differ from "testCase_a". However, the execution trace of "testCase_b" is $testCase\_b < m1, m2, m3 >$, which is very similar (in this scenario same) to the execution trace of "testCase_a", in terms of the method call sequence.

To quantify the similarity between a new/modified test case and a failing test case from history, test cases can be represented by their sequences of method calls as described in the previous example (Fig. 4.3). The sequence of method calls could be

```
testLang747()
NumberUtils.createNumber(java.lang.String)
StringUtils.isBlank(java.lang.CharSequence)
NumberUtils.createInteger(java.lang.String)
. . . . .
StringUtils.isBlank(java.lang.CharSequence)
NumberUtils.isAllZeros(java.lang.String)
. . . . . .
.. . . . . .
StringUtils.isBlank(java.lang.CharSequence)
NumberUtils.createInteger(java.lang.String)
```

(a) Test case trace of Fig. 4.1

```
testStringCreateNumberEnsureNoPrecisionLoss()
math.NumberUtils.createNumber(java.lang.String)
StringUtils.isBlank(java.lang.CharSequence)
NumberUtils.isAllZeros(java.lang.String)
NumberUtils.createFloat(java.lang.String)
. . . . . .
. . . . . .
NumberUtils.createNumber(java.lang.String)
StringUtils.isBlank(java.lang.CharSequence)
NumberUtils.isAllZeros(java.lang.String)
```

(b) Test case trace of Fig. 4.2

Figure 4.4: Example of execution traces for Fig. 4.1 and Fig. 4.2

extracted from their execution traces. For instance, the sequences for the two test cases of Fig. 4.1 and Fig. 4.2 are shown in Fig. 4.4a and Fig. 4.4b. As it can be seen, $NumberUtils.createNumber(java.lang.String)$, $StringUtils.isBlank(java.lang.CharSequence)$ and *NumberUtils.isAllZeros(java.lang.String)* methods are the same in the two test case traces, which makes the two test cases similar.

This example, and other cases like this, which show the failing test case trace in the current release is very similar to the failing test case traces from previous releases, are the motivations behind this thesis, where I have proposed a quality metric using test case similarity to the historical failing test cases.

4.2 High-level Overview of the Proposed Approach

My proposed similarity-based metric considers a test as effective, if it is similar to any of the failed test cases from the previous versions. The similarity between test cases is defined based on their sequences of method calls, extracted from execution

traces. Therefore, first, execution traces containing the sequence of method calls need to be retrieved for all previously failed test cases. Then, the sequences of method calls (i.e., execution traces) are also collected for all modified or newly added tests in the current version. Finally, a similarity function is used to determine the similarity between the execution traces of the modified/new tests in the current version and the previously failed tests. Hence, the input of the similarity function is the execution traces of both the current releases' modified/new tests and the previously failed tests. The similarity function, at the end, returns a value indicating how similar is a test case to the previously failed test cases.

There are several similarity functions that can be used to identify similarity between two inputs/sequences. The main difference between these functions is whether they account for the order of the method calls in the sequence or not (i.e., they take it as a Set or a Sequence). In my thesis, I use the following similarity functions to implement the similarity-based test quality metric.

4.2.1 Set-based Similarity Functions

In this subsection, I explain two set-based similarity functions that I have used in this thesis.

Basic Counting (BC):

This measure is a very basic function, which does not account for the method call orders nor for their position in the sequence. The function simply looks at the past failing sequences of unique method calls and counts the overlap with the unique

method calls of the test case under study. The total similarity value of a test case is the sum of all these occurrences, which indicates how many unique method calls from the current test case also appeared in the previous failing tests traces. However, instead of using the actual summation value, a normalized value between 0 and 1 is used. The normalization has been performed by diving the total summation value by the total number of unique method calls from the history. The higher normalized value of a test case means the test has also higher similarity with the previous failing tests.

Hamming Distance (HD)

Hamming Distance is a widely used distance functions used in the literature, which is a basic edit-distance. The edit-distance between two sequences is defined as the minimum number of edit operations (insertions, deletions, and substitutions) needed to transform the first sequence into the second [Dong and Pei, 2007; Gusfield, 1997]. Hamming is only applicable on identical length inputs and is equal to the number of substitutions required in one input to become the second one [Dong and Pei, 2007]. If all inputs are originally of identical length, the function can be used as a sequence-aware measure. However, in most of the applications, test inputs have different lengths. Therefore, to force them to have an identical length, a binary vector is made per input that indicates which elements from the set of all possible elements of the encoding exist in the input. As a result, the function does not preserve the original order of elements in the input anymore and it becomes a sequence ignorant (or set-based) similarity function [Hemmati and Briand, 2010; Hemmati et al., 2013].

In my study, the total hamming distance of a modified/new test, e.g., $T1$ in the current release, has been calculated by summing up all hamming distances, between $T1$ and each of the failing tests from the previous releases. For example, assume, two previous failed test traces are $T3 < A(), B(), B(), C(), B(), D() >$ and $T2 < A(), E(), D(), B(), A() >$, and the modified/new test trace in the current release is $T1 < F(), A(), C(), M(), N(), X(), A() >$. Therefore, the total hamming distance of $T1$ would be the summation of hamming distances between $T1$ and $T3$ (i.e., *Hamm(*$T1, T3$*)*), and hamming distances between $T1$ and $T2$ (i.e., *Hamm(*$T1, T2$*)*).

To calculate the hamming distance between $T1$ and $T2$ (*Hamm(*$T1, T2$*)*), first, all unique method calls from $T1$ and $T2$ form a set of all possible elements, i.e., $V < A(), E(), D(), B(), F(), C(), M(), N(), X() >$. Now, both the $T1$ and $T2$ are encoded as binary vector of identical length, where a bit is true only if the encoded test case contains the corresponding element from V. So, the test $T1$ is encoded as $< 1, 0, 0, 0, 1, 1, 1, 1, 1 >$ and $T2$ is encoded as $< 1, 1, 1, 1, 0, 0, 0, 0, 0 >$, with respect to the set of all possible elements V.

Then *Hamm(*$T1$*,* $T2$*)* has been 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. Therefore, the normalized *Hamm(*$T1$*,* $T2$*)* is $< 0, 1, 1, 1, 1, 1, 1, 1, 1 >= 8/9$. Similarly, the normalized hamming distance between $T1$ and $T3$ (*Hamm(*$T1, T3$*)*) has also been calculated.

Finally, all of these normalized hamming distances have been summed up and normalized between 0 and 1 again by dividing the summation value by the total number of failed test cases from previous versions. The low hamming distance value of

a test case means the test has high similarity with the previous failing tests. So, I have converted this distance to similarity by subtracting the total normalized hamming distance value from 1.

Improved Basic Counting (IBC)

I propose 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 (when it is higher than the threshold). For example, the similarity value of a test case TC_x using BC and HD is 0.4 and 0.6 respectively. Now assume the threshold similarity value for IBC is 0.5. So the IBC value of TC_x would be 0.6 in this case. Thus HD will be used as IBC similarity value only when the BC is lower but HD is higher than the threshold value. Otherwise BC will be used as the default IBC similarity value.

4.2.2 Sequence-based Similarity Functions

In this thesis, I have used the following sequence-based similarity function for the proposed similarity-based metric.

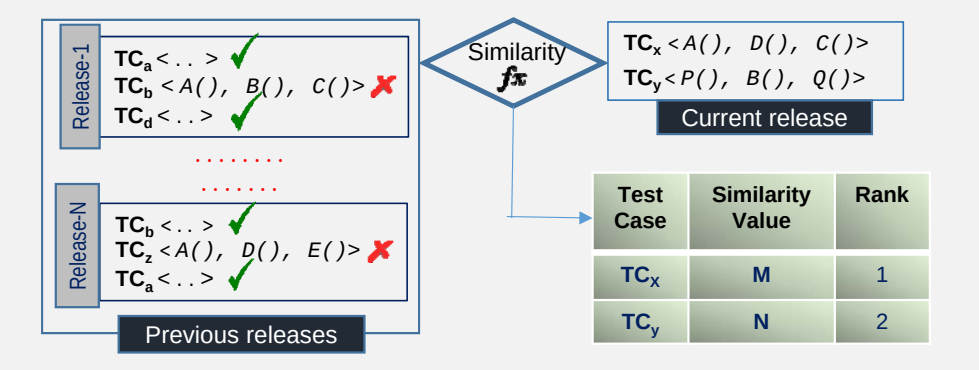

Figure 4.5: An overview of the proposed similarity-based test case quality measure, using historical failure data.

Edit Distance (ED)

The general edit distance function is a sequence-aware function, where the order and position of method calls in the traces would matter. One of the most well-known algorithms implementing edit-distance which is not limited to identical length sequences is Levenshtein [Dong and Pei, 2007] where each mismatch (substitutions) or gap (insertion/deletion) increases the distance by one unit. To change distances into similarities, we need to reward each match and penalize each mismatch and gap. The relative scores assigned to matches, mismatches, and gaps can be different. A basic setting for the function would be implemented in a way where matches are rewarded by one point and mismatch and gap are treated the same by giving no reward [Hemmati and Briand, 2010; Hemmati et al., 2013]. In my study, I have used this basic setting for the implementation.

4.3 Similarity-based Quality Measure

The proposed similarity-based quality metric measures tests case quality by their similarity values, which are calculated using different approaches mentioned in the previous sections. The test cases are ranked in descending order based on the similarity value. The lower rank value of a test indicates that the test case should be executed earlier than the other tests and is more effective in identifying the first fault of a system. The overall process of similarity based quality measure using BC similarity function is shown in Fig. 4.5 where the previous failed tests are TC_b and TC_z, and the modified/added tests in the current release are TC_x and TC_y. The higher rank of TC_x indicates that it should be executed earlier than TC_y to detect the faults earlier.

Chapter 5

Evaluation

In this chapter, I explain detailed empirical study and evaluate the results of my experiments. I evaluate the proposed quality metrics in the context of tests case prioritization that is one of the application domains of such quality metrics.

5.1 Research Questions

In this thesis, I seek answer to the following research questions:

RQ1: Can a similarity-based test case quality metric improve the traditional historical fault detection metric, in terms of identifying fault revealing test cases?

RQ2: Which similarity function works best for the similarity-based test quality metric?

RQ3: Can the best similarity-based test quality metric improve other existing quality metrics, such as code coverage, test size and change-related metrics?

Table 5.1: 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	27	2,245	1,757
Joda Time	65	4,130	3,748

RQ4: How well can we predict the first failing test when combining the similarity based metrics with the traditional metrics?

RQ5: Overall, can the best prediction-based quality metric improve the best individual quality metric?

5.2 Subjects Under Study

In my experiment, I have used five different Java projects from the *defects4j* database [defects4j]. The database provides 357 faults and 20,109 Junit tests from five different open-source Java projects as mentioned in Table 5.1. All the faults are real, reproducible and have been isolated in different versions [Just et al., 2014]. There is a faulty version and a fixed version of the program source code, for each fault. The faulty source code is modified in the fixed version to remove the fault. The test cases are the same in both of the faulty and the fixed versions. However, there is at least one test case (a Junit test method) in each version that fails on the faulty version but passes on the fixed version.

It is worth mentioning that I am not focusing on specific types of faults like memory issues, recursions, concurrency or timing issues in my thesis. Instead, I propose a

generic approach (that is mainly built upon existing risk-based approaches) for typical faults in typical deterministic systems, regardless of their application domain. My case studies using *defects4j* database [defects4j], however, only focus on a small subset of open source java libraries. Indeed, generalizing the results requires replications on different systems, which is not within the scope of my thesis.

5.3 Data Collection

As my proposed similarity functions require test case traces containing method call sequences, I need to use a tool for trace generation. I have used Daikon [Ernst et al., 2007] tool to produce the execution traces from three projects (Commons Lang, Joda Time and JFreeChart). Daikon is a tool to dynamically detect likely program invariants and it allows to detect properties in C, C++, C#, Eiffel, F#, Java, Perl, and Visual Basic programs [Ernst et al., 2007]. The daikon front end (instrumenters) for Java, named Chicory, executes the target Java programs and creates the *.dtrace* file that contains the program execution flow along with the variable values in each program point. The method sequence calls have been extracted from the *.dtrace* file.

However, for the other two projects (Commons Math and Closure Compiler), I could not use daikon for trace collection, as the tests from these projects generated very large *.dtrace* files (more than 5 GB for each test case). So, instead of using daikon, I have used AspectJ [Eclipse-AspectJ] to produce the trace of method sequence calls directly. Although the trace extraction process is different, the format of the extracted method sequence calls is same.

I could have used AspectJ for all of the projects, however, I actually started working with Daikon initially. When I faced problem with the large Daikon trace files for two of the projects, I started using AspectJ. As the extracted method sequence format is exactly the same for both the Daikon ad AspectJ, I did not repeat the extraction process using AspectJ for other three projects for which I used Daikon.

I have collected the method sequences for all modified tests in the current release and also all the failed tests traces from the previous releases. Finally, I have measured the tests quality in each version based on their ranks provided by different approaches for the evaluation.

Moreover, I have also extracted some basic quality measures, such as, Size of Testcase (ST), Method Coverage (MC) and Changed Method Coverage (CMC) from the projects for the evaluation. The ST is the number of uncommented statements in a test method that has been derived from the Java Abstract Syntax Tree (AST) parser using Eclipse JDT API [Eclipse-JDT]. The MC of a test case is the number of unique methods called during the test execution. I have calculated this measure from the execution traces produced by daikon and AspectJ. The 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. At first, I have identified the list of source code method names in the current version that have been changed from the immediate previous version. Then I have measured how many of these changed source code methods appear in the execution trace of each modified/new test cases in 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.

I have implemented the similarity functions using Java and I have used the statistical computing tool, R [R-Project] for my evaluation purposes. R is a language and environment that provides a wide variety of statistical computing technique. Another reason behind using R is that it is a free software and a number of libraries are available for statistical computation.

5.4 Experiment Design

Experiment Design for RQ1 (Can a similarity-based test case quality metric improve the traditional historical fault detection metric, in terms of identifying fault revealing test cases?): To answer RQ1, I have compared the test cases ranks using the proposed similarity-based metric using Hamming Distance (HD) function with the traditional history-based metric (TM).

Experiment Design for RQ2 (Which similarity function works best for the similarity-based test quality metric?): To answer RQ2, I have compared HD, Basic Counting (BC), and Edit Distance (ED) functions by analyzing the ranks of test cases provided by the similarity-based metrics using these functions.

I have also proposed an Improved Basic Counting (IBC) metric. There are two improvements in the IBC compared to the BC. In the improved BC, at first, the normalized BC and HD values have been rounded by two decimal places. This helps to avoid ranking differences where the similarities are very close. The second improvement is to combine the BC and HD. To do that for cases where BC values are lower

than 0.5, I have first looked at HD values if they are higher than 0.5. In that case, I have ranked them using HD, otherwise ranked them using BC. This helps to get the most out of both BC and HD. IBC similarity function has been actually designed based on the initial performance of BC and HD. Although I use both the BC and HD to design IBC, I give more weight to the BC, as BC was found slightly better than the HD in identifying the first faults.

Experiment Design for RQ3 (Can the best similarity-based test quality metric improve other existing quality metrics, such as code coverage, test size and change-related metrics?): To answer RQ3, I have compared the Size of Testcase (ST), Method Coverage (MC) and Changed Method Coverage (CMC) metrics against the proposed similarity-based metric.

Experiment Design for RQ4 (How well can we predict the first failing test when combining the similarity based metrics with the traditional metrics?): RQ4 compares the ranks of test cases using two prediction models, 1) the traditional model and 2) my proposed model. In the traditional model, all the traditional metrics such as ST, MC, CMC and TM have been used to build a regression model, where the metrics are used as independent variables. On the other hand, the proposed model uses all the similarity-based metrics using HD, BC, ED and IBC functions, along with all the traditional metrics.

To predict the ranks of the test cases in the current version of a project, the regression models had been trained using different quality measures from all the previous versions. For example, to rank the tests in a latest version (e.g., version

10) of a project, the model had been trained using all quality measure values from version 1 to 9. The test cases data of the current version have been used as the test set of the model, where ranks of the tests have been predicted using the model. It is worth mentioning that a test case can either pass or fail in the actual execution. Therefore, the dependent variable of the regression model is binary (pass or fail), which makes the model as a logistic regression model. I have used the "glm" function in R [GLM] to build the logistic regression model. The "glm" function builds the logistic regression model using a set of predictor variables and the model is used to predict a binary outcome (i.e., pass or fail). Then, I have used "predict" function [Predict] in the R tool to get the response probability value (i.e., probability of a test being passed/failed, in my case) of the observations from the test set.

Experiment Design for RQ5 (Overall, can the best prediction-based quality metric improve the best individual quality metric?): To answer RQ5, I have compared the ranks of the test cases using the best individual quality metric with the ranks of the test cases using the best prediction model.

5.5 Evaluation Metric

The defects4j dataset mentions which test case actually fails in the current version [defects4j]. So, I have compared the first failing test case's rank, provided by different approaches for each version (note that there is only one fault per version, but there might be more than one test that detect it). In other words, I have compared the percentage of the test cases that need to be executed in order to catch the first fault

in each version, separately. I have done this by dividing the rank of the first failing test by the total number of modified/new tests in each version. In *defects4j* dataset, most of the versions contain only one failing test case (i.e., one failure only). So to detect the failure, even one failing test is enough. Therefore, I have used the rank of the first failing test to compare different prioritization approaches.

In case of ties in the similarity values, I have ranked the tied test cases randomly, by applying the *rank* function in R with a parameter *ties.method="random"* [R:Ranks]. To deal with this randomness, I have calculated the rank 30 times for each version of the projects. Running an experiment 30 times is a common practice to deal with the randomness in different fields of sciences [Arcuri and Briand, 2011]. Therefore, I have 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 I have calculated the rank 30 times for each version, I have considered the median of these 30 values as the rank of first failing test for each version. Finally, I have 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. A boxplot is a convenient way to represent the distribution of data through quartiles. It shows five number summary: minimum, first quartile, median, third quartile, and maximum. Fig. 5.1 shows a simple boxplot that represents uniformly distributed data between 0 and 1.

Whenever I compare two distributions of the results and say one outperforms the other, I have first conducted a non-parametric statistical significance test (U-test) [Mann and Whitney, 1947] to make sure the differences are not due to the randomness

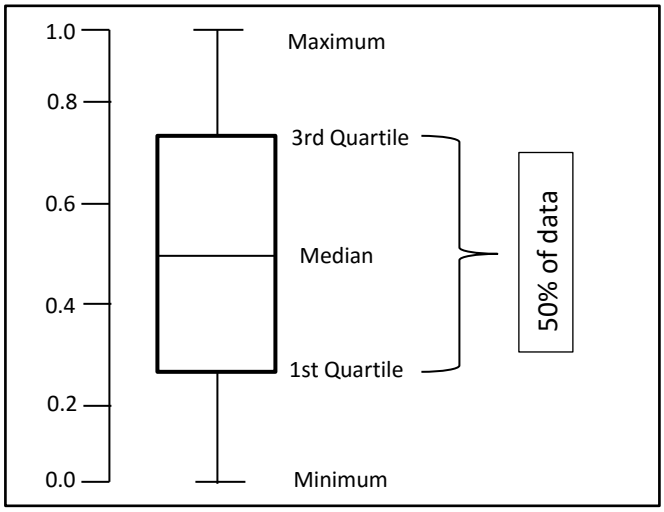

Figure 5.1: A simple boxplot

of the algorithms. I have reported the p-values from the U-test to represent the statistical significance. It is considered when p-value is less than 0.05, the results are statistically significant and are not due to randomness [Mann and Whitney, 1947].

Sometimes, the difference between the performance of two algorithms are statistically significant (using U-test and p-value), however, the difference might be so small that it has no practical value. Therefore, it is crucial to assess the magnitude of the differences as well [Arcuri and Briand, 2011]. Effect size measures are used to analyze such a property [Arcuri and Briand, 2011; Goulden, 2006]. In my study, I have used a non-parametric effect size measure, called Vargha and Delaneys $\hat{A}_{12}$ statistics [Arcuri and Briand, 2011; Goulden, 2006; Vargha and Delaney, 2000]. 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 Y, $\hat{A}_{12} = 0.7$ represents that we would obtain higher results 70% of the time

Table 5.2: 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

with algorithm X compared to the algorithm Y [Arcuri and Briand, 2011].

In my 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 an $\hat{A}_{12}$ value for each version of the project while comparing two prioritization approaches. So, in our case, $\hat{A}_{12} = 0.8$ indicates that algorithm X ranks the first failing test lower than algorithm Y 80% of the time. In other words, algorithm X can detect the fault faster than algorithm Y 80% of time. I have used this $\hat{A}_{12}$ measure to evaluate all of my results.

It is worth mentioning that I have considered only the versions with at least 5 modified/new test cases for the evaluation. Actually, fewer than 5 modified/new test cases in a version is too small to run the prioritization. So, we assume that in the versions having fewer than 5 modified/new test cases, executing all test cases is not costly and hence the prioritization is not actually beneficial. The details of my number of studied versions is mentioned in table 5.2.

Table 5.3: Comparison of the first failing test's rank using TM and HD

Projects	Measure	HD	TM
JFreeChart	Mean Rank	0.5385	0.6154
	Std. Deviation	0.1087	0.5439
Commons Lang	Mean Rank	0.4205	0.4645
	Std. Deviation	0.2421	0.3151
Commons Math	Mean Rank	0.4753	0.5283
	Std. Deviation	0.1739	0.3512

5.6 Results and Discussion

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

5.6.1 Experimental Results for RQ1

RQ1: Can a similarity-based test case quality metric improve the traditional historical fault detection metric, in terms of identifying fault revealing test cases?

To answer RQ1, I have compared the first failing test's rank assigned by my proposed similarity-based metric (in this case, by using the Hamming Distance (HD) function) and the traditional historical fault-based metric. The traditional metric ranks the previous failing tests higher in the current release. Therefore, the traditional approach works only when the set of all modified/new tests in the current release contains at least one test that failed in any of the earlier versions.

Table 5.2 shows the number of versions where the traditional metric (TM) works. As can be seen from the table, TM works only in 2, 12 and 20 versions, respectively

from the JFreeChart, Commons Math and Commons Lang projects. Moreover, TM does not work in any of the versions from the Closure Compiler and Joda Time projects. The large number of versions where the traditional approach is not working at all, is indeed the main motivation behind this thesis to propose an improved historical fault-based metric. Since the traditional metric can not rank the tests in many versions, it is already falling behind the proposed metric, but to be fair, I have also compared the results for only the working versions.

For these versions of the projects, I have compared the rank of the first failing test using the traditional metric against the similarity-based metric (in this case, by using HD as the similarity function). Table 5.3 compares the average rank of the first failing tests of the working versions from project JFreeChart, Commons Lang and Commons Math. It can be observed that the average rank using HD is lower and has lower variance than TM. The lower average rank of the first failing test suggests that if the tests are executed based on the ranks provided by the HD, the fault would be detected faster than using the TM-based rank.

I have also used the $\hat{A}_{12}$ measure to compare these two metrics. Fig. 5.2 shows the boxplots of the $\hat{A}_{12}$ measures distribution for the versions of three projects where the traditional metric works. The higher than 0.5 median lines in the boxplots represent that in general ranks provided by the similarity-based metric using HD similarity function is better than the ranks using TM, even for cases where the traditional metric is working (note that the differences are also statistically significant with p-values<0.05).

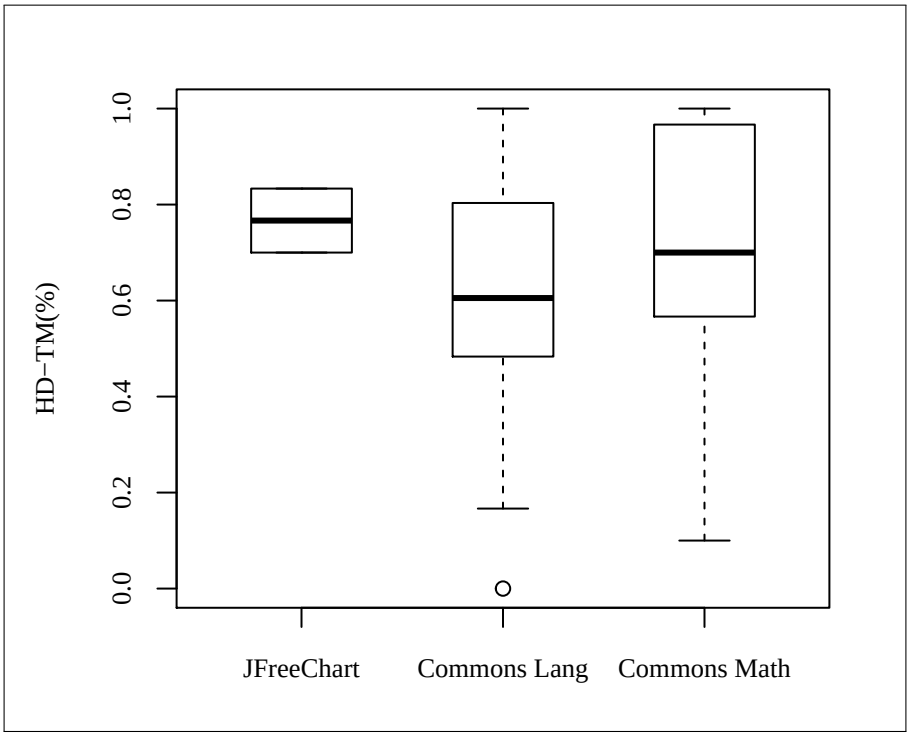

Figure 5.2: The boxplots of the effect size measures for finding the first fault using HD and TM, when comparing 30 runs of each versions of each project.

5.6.2 Experimental Results for RQ2

RQ2: Which similarity function works best for the similarity-based test quality metric?

To answer RQ2, I have compared the Basic Counting (BC), Hamming Distance (HD), Edit Distance (ED) and Improved BC (IBC) similarity functions. I have looked at the ranks of the first failing tests provided by the proposed metric using these similarity functions. Moreover, I have checked the $\hat{A}_{12}$ measure distributions for this evaluation.

The boxplots in Fig. 5.3 shows the distribution of the first failing tests ranks for all the versions of five projects using different similarity functions such as ED,

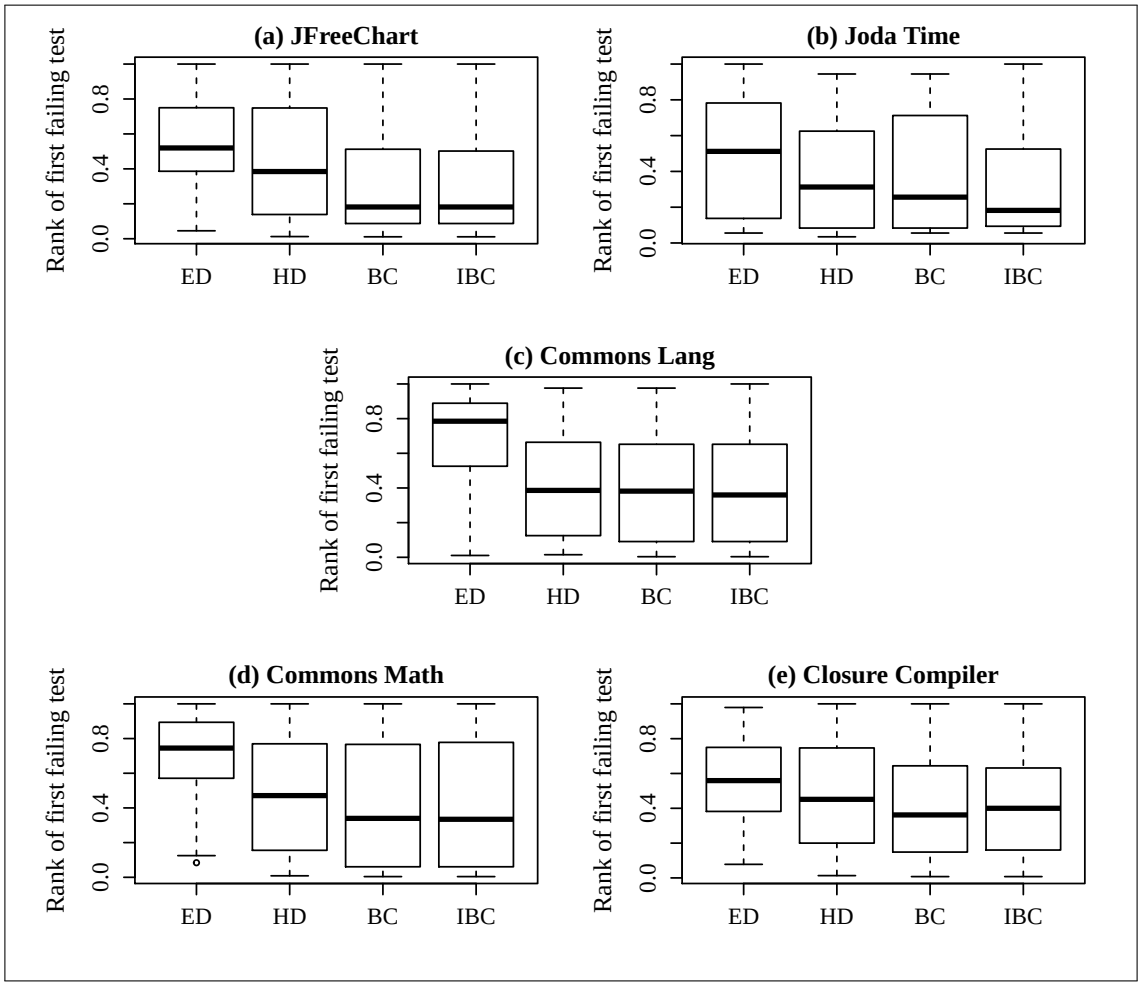

Figure 5.3: The boxplots of the average first failing test's rank using HD, ED, BC and IBC for all versions of each project.

HD, BC and IBC. It is seen that the median rank of the first failing test using the sequence-based similarity function (i.e., ED) is always higher than the other set-based similarity functions (i.e., HD, BC and IBC) for all the projects. As the lower rank of the first failing test represents earlier fault detection, the lower median line of a boxplot represents better ranking. Therefore, on average, ED is falling behind the other similarity functions in all projects. Similarly, on average, the ranks using BC

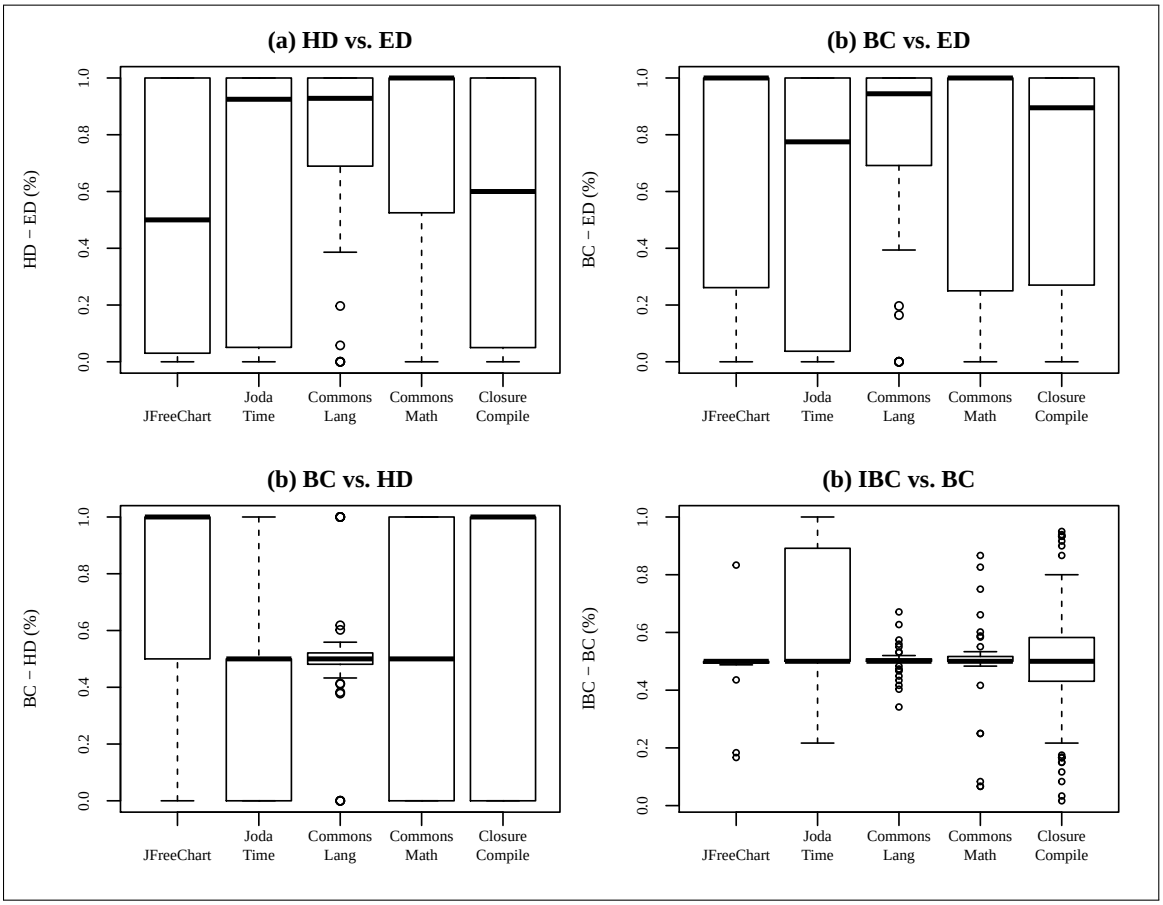

Figure 5.4: The boxplots of the effect size measures for finding the first fault using BC, HD, ED, and IBC, when comparing 30 runs of each versions of each project.

and IBC is better than the use of HD. However, while comparing the BC and IBC, the performance looks pretty similar for all the projects except the Joda Time (Fig. 5.3b) where IBC is better than BC.

Besides comparing the distribution of the first failing test cases ranks of a project using different approaches, I have also measured their magnitude of differences by using $\hat{A}_{12}$ measures. Note that I calculate the ranks 30 times using an approach and I obtain one $\hat{A}_{12}$ measure value for each version while comparing two approaches in a project. Fig. 5.4 shows boxplots of the $\hat{A}_{12}$ measures distribution using different

approach for each project, where there are at least 5 new/modified test cases in a version.

Fig. 5.4(a) compares HD and ED, and Fig. 5.4(b) compares BC and ED. It can be observed that the similarity-based ranking using HD provides significantly better result than the use of ED in Joda Time, Commons Lang and Commons Math projects, as the median lines for these box plots are closer to 1.0. However, for the remaining two projects, the improvements are not enough significant (p-values are still lower than 0.05 but the median effect size is around 50%). On the other hand, BC is much better than the ED for all of the projects. Therefore, the results shown in Fig. 5.4(a) and Fig. 5.4(b) suggests that it is better to use the BC as a similarity function compared to the ED. The results also represent that the sequence-aware similarity function (ED) is falling behind the sequence-ignorant (or set based) similarity function (BC) in test case prioritization. This result is actually inline with the recent studies [Hemmati et al., 2013] in the context of test case selection/prioritization. One plausible explanation is that the sequence-aware matching is too restrictive, which ignores potential weak similarities

Next, I have compared the similarity function BC with HD and the results are shown in Fig. 5.4(c). Here it can be seen that BC performs significantly better than the HD in JFreeChart and Closure Compiler project. However, the performance of BC and HD is quite similar for the Commons Lang and Commons Math projects, where neither of these functions outperforms the another (again p-values are low but the median effect size is around 50%). On the other hand, HD performs better than the BC for the Joda Time project. This result actually motivated me to propose the

improved similarity function, IBC (defined in the section 5.4, which uses both the BC and HD values. However, I give more weight to the BC values than HD values to calculate the IBC, as the BC ranks are slightly better than the HD ones in identifying the first fault in the current version.

Fig. 5.4(d) shows the comparison between IBC and BC. As can be seen from the figure, IBC improves BC in the Joda Time project (p-values$<$0.05 and the distribution of effect sizes leaning more toward higher than 50%) and is as good as BC in the others. Therefore, IBC appears to outperform the other metrics based on the results of my studied projects.

5.6.3 Experimental Results for RQ3

RQ3: Can the best similarity-based test quality metric improve other existing quality metrics, such as code coverage, test size and change-related metrics?

To evaluate RQ3, I have compared the ranks using the best similarity function (in this case IBC, based on RQ2 results) with the other traditional measures, e.g., Traditional Metric (TM), Method Coverage (MC), Size of Testcase (ST) and Changed Method Coverage (CMC). These other traditional metrics have been explained in chapter 2 and Section 5.3. To evaluate the results of RQ3, I have also used the distribution of the first failing test case rank for all versions of a project. Moreover, I have used the $\hat{A}_{12}$ measure distribution in a boxplot to compare two metrics and again I have considered the versions having at least 5 new/modified test cases for all projects.

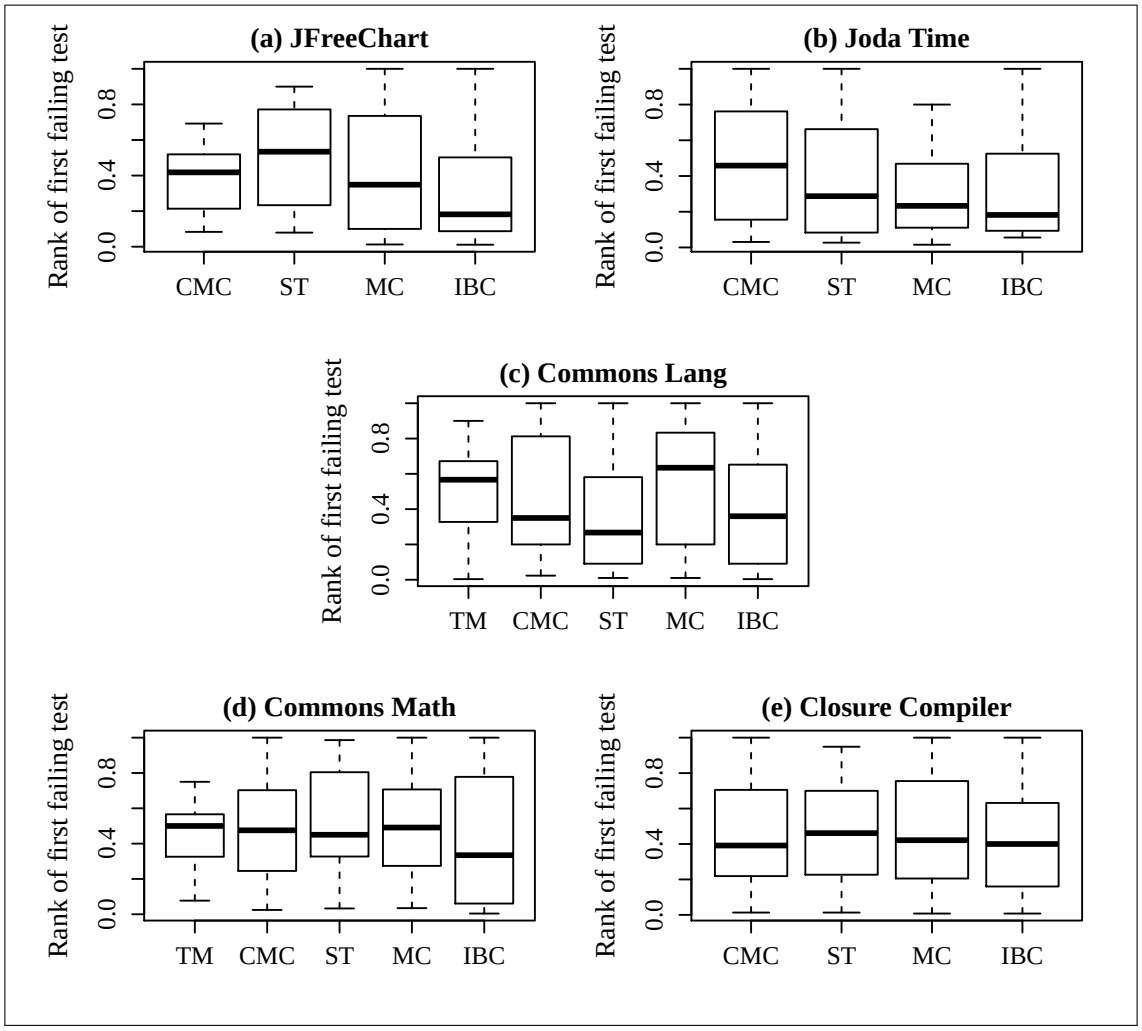

Figure 5.5: The boxplots of the average first failing test's rank using TM, CMC, MC, ST, and IBC for all versions of each project.

Fig. 5.5 shows the distribution of the first failing tests ranks using different traditional metrics and IBC. Note that TM does not work in any version of the Closure Compiler and Joda Time project (as mentioned in section 5.6.1 and Table 5.2). In addition, in the JFreeChart project, TM works only for 2 versions. Therefore, I have excluded these 3 projects while comparing their performance using TM with the other metrics. It can be observed that the median line for IBC is lower than the other met-

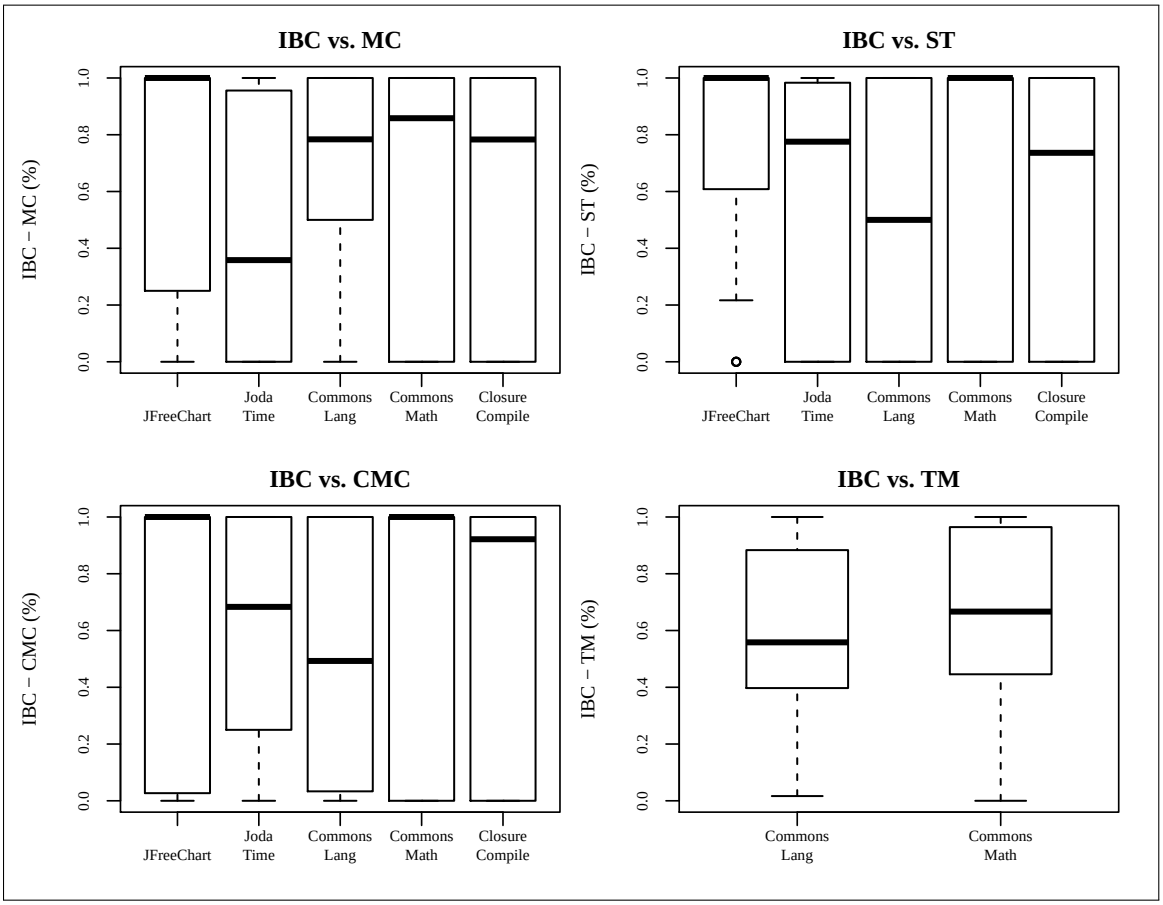

Figure 5.6: The boxplots of the effect size measures for finding the first fault using IBC, CMC, ST, MC and TM when comparing 30 runs of each versions of each project.

rics in almost all the projects. The only exception appears in the Commons Lang project (Fig. 5.5c) where the median line of IBC is slightly higher than the ST.

The performance of the best similarity metric (IBC) against CMC, ST, MC and TM (applicable in 2 projects) using $\hat{A}_{12}$ is shown in Fig. 5.6. In general, IBC ranks the failing test lower than any of these metrics in more than 80% of the cases for three projects (JFreeChart, Commons Math and Closure Compiler). Also in Commons Lang project, IBC is significantly better than the MC, in 80% of the cases. However,

the differences are not very significant (median effect size around 55-60% with p-values <0.05) when comparing IBC with ST or CMC in Commons Lang and Joda Time projects. Nonetheless, there is only one case that IBC falls behind and that is in JodaTime when comparing IBC and MC, where the median effect size is around 40%. The results also show that neither of these traditional metrics completely outperforms others for all the projects. However, the proposed IBC is much more consistent in all the projects.

5.6.4 Experimental Results for RQ4

RQ4: How well can we predict the first failing test when combining the similarity based metrics with the traditional metrics?

To answer RQ4, I have compared the ranking using predicted ranks from two different logistic regression models. In the regression model using traditional metrics, I have considered all the traditional metrics (e.g., TM, ST, MC and CMC) as the predictors (i.e., independent variables) and in the proposed regression model, I append the list of traditional predictors by adding the similarity-based metrics (e.g., ED, HD, BC and IBC).

However, there might be several variables in the model that are highly correlated with each other. In addition, all the variables might not be statistically significant for the prediction model. To deal with these problems, I have removed highly correlated variables (i.e., any variables with correlation higher than 0.5) as suggested by Shihab et al. [2011]. I 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

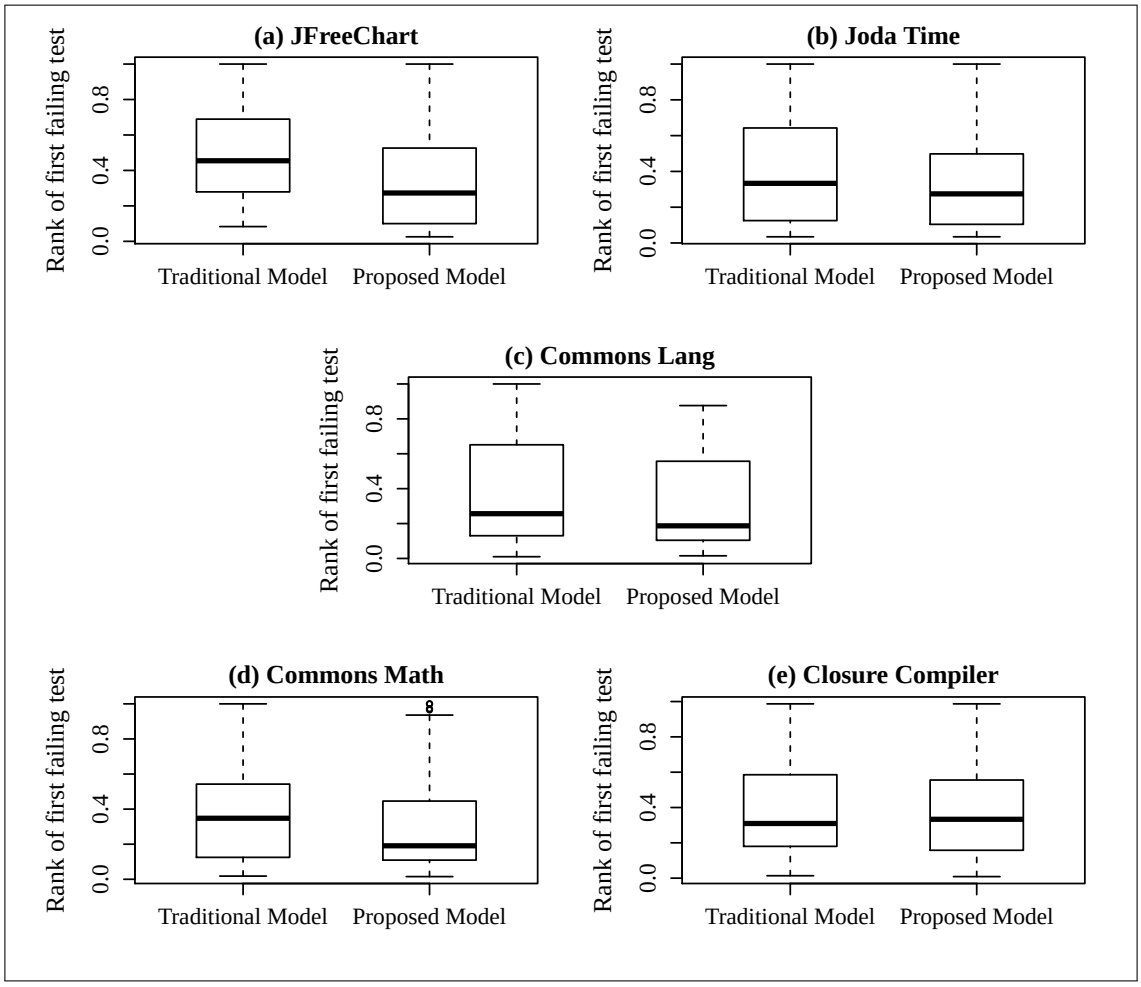

Figure 5.7: The boxplots of the average rank of the first failing test rank using two prediction models for all versions of each project (Traditional Model: uses all traditional metrics - Proposed Model: uses all traditional and similarity-based metrics).

by previous works [Shihab et al., 2011; Cataldo et al., 2009]. The VIF quantifies the severity of multicolinearity in an ordinary regression analysis where lower VIF (=1) represents the variables used in the regression analysis are not correlated with each other [Cataldo et al., 2009]. Thus, the prediction model is built with the least number of uncorrelated factors. In addition, I have also measured the p-value of

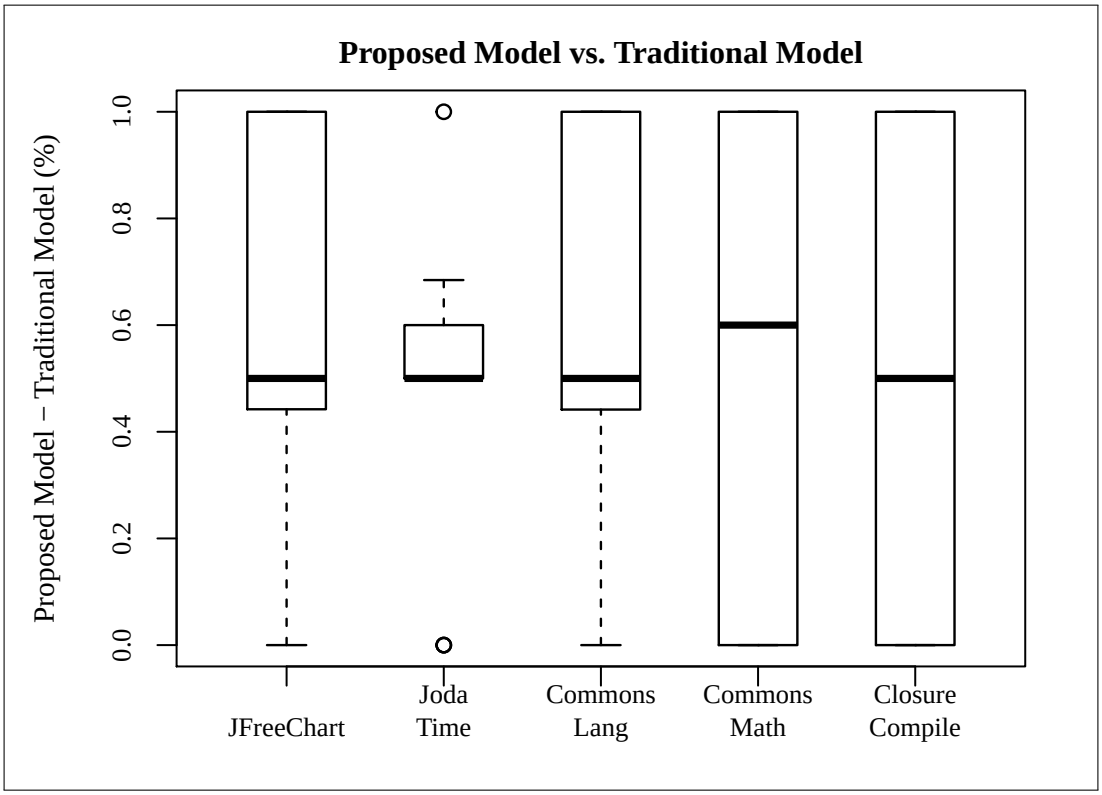

Figure 5.8: The boxplots of the effect size measures for finding the first fault using the two prediction models, when comparing 30 runs of each versions of each project.

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.

Fig. 5.7 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, I 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,

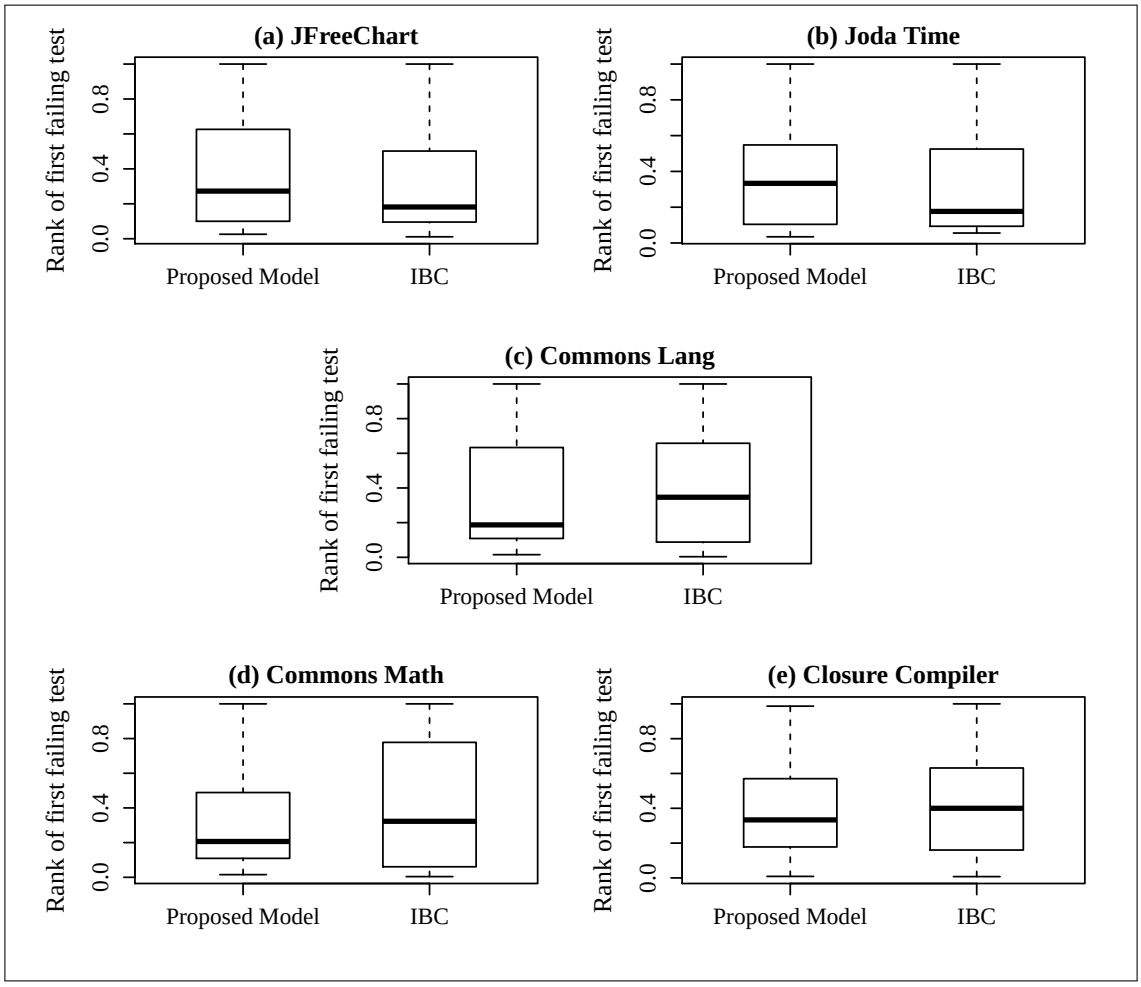

Figure 5.9: The boxplots of the average rank of first failing test using my proposed prediction model and IBC for all versions of each project.

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 Fig. 5.8 that compares the $\hat{A}_{12}$ measure distribution of the first failing tests ranks from the two regression models.

5.6.5 Experimental Results for RQ5

RQ5: Overall, can the best prediction-based quality metric improve the best individual quality metric?

RQ5 compares the predicted ranks from the proposed logistic regression model (as it is better than the traditional model, based on RQ4) with the ranks using the best-similarity metric that is IBC (based on RQ2 and RQ3). Fig. 5.9 shows distribution of the first failing test's rank using prediction model and IBC for all versions of the projects. Besides, the $\hat{A}_{12}$ measure distribution comparing these two approaches is also shown in Fig. 5.10.

It is observed that IBC performs better than the predicted ranks for the JFreechart and Joda Time project, as the median line is below 0.5 in Fig. 5.10 and also it is lower than the prediction model-based ranks in Fig. 5.9. However, the opposite behavior is observed for the other 3 projects, Commons Lang, Commons Math and Closure Compiler, where the model-based ranks are better than the ranks using IBC. One plausible reason would be the higher amount of historical data has been used to train the regression model for these 3 projects, which provides better ranks than the use of individual IBC. Note that Commons Lang, Commons Math and Closure Compiler projects have data from more versions (41, 36 and 85 versions respectively, mentioned in Table 5.2) compared to the JFreeChart and Joda Time projects (16 versions each). Therefore, the results suggest to use ranks from the prediction model when the model could be trained using enough data, and to use IBC otherwise.

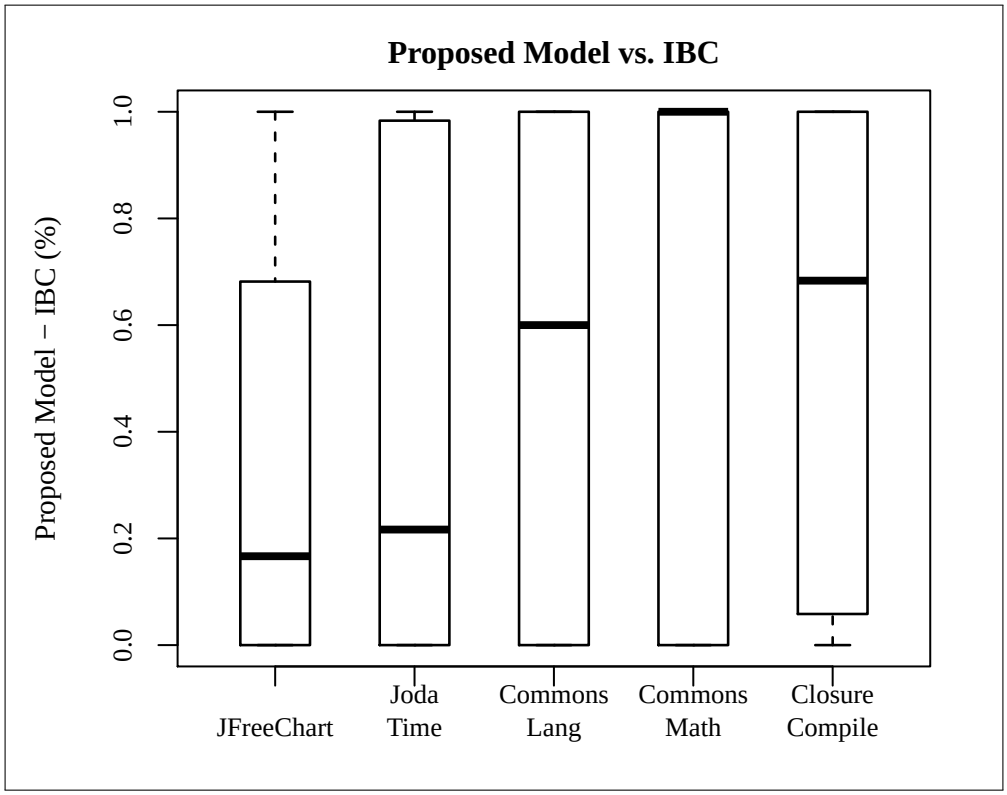

Figure 5.10: The boxplots of the effect size measures for finding the first fault using my proposed prediction model and IBC, when comparing 30 runs of each versions of each project.

5.7 Threats to Validity

In terms of conclusion validity, I 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 [Feldt and Magazinius, 2010], I have used existing libraries and tools as much as possible (e.g., Daikon [Ernst et al., 2007], AspectJ [Eclipse-AspectJ]). However, one potential threat would be the impact of "method name change" in the new releases, when the body of the method does not change.

In such cases, my method sequence-based similarity function is weak. In general, if one observes significant cases of "method name change" in the history, a finer grain representation of test cases (e.g., sequences of covered statements) is recommended. However, in my case studies, I did not experience any case where only method name changes (the total "method name change" statistics is 4 versions out of 227 versions=1.7%).

In terms of construct validity [Feldt and Magazinius, 2010], I have used 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) [Rothermel et al., 1999], which is commonly used for test prioritization evaluation. However, in my study, each version contains only one fault. So, using APFD would be ambiguous.

In terms of external validity [Feldt and Magazinius, 2010], I have conducted the empirical study based on five real-world open source java libraries from *defects4j* database with several versions and faults. However, generalizing the results to different types of systems may still require further experiments.

5.8 Summary

In this chapter, I have explained a detailed empirical study and evaluated the experimental results based on my research questions. I have used several versions of five different java projects from defects4j database [defects4j] for my case study. The rank of the first failing test case for each version has been used for evaluating different quality metrics. Each experiment has been performed 30 times to deal with the randomness and the median rank of these 30 runs has been reported.

In addition to reporting the boxplot distribution of median ranks of first failing test cases from several versions, I have also reported the effect size ($\hat{A}_{12}$ measure) while comparing two approaches. I have also ensured the results are statistically significant (p-value is less than 0.05) and p-values are reported accordingly.

The experimental results show that on average, the use of similarity-based test case quality metric using historical failure data can identify the first fault earlier than using the existing quality metrics (the only exception is in the Commons Lang project, where the existing metric ST was found better). I have also compared different similarity functions and found that the Improved Basic Counting (IBC) metric outperforms the other similarity functions (e.g., ED, HD, BC) in identifying the first fault. Therefore, I propose to use the IBC similarity function to implement the similarity metric. Moreover, the predicted ranks provided by the logistic regression model using combined set of metrics have been compared against the ranks using the best individual similarity-based metric. The results suggest that it is better to use the combined set of metrics in the prediction model when a larger training data from previous versions is available for a project.

Chapter 6

Conclusion

In the applications such as test case prioritization, generation, selection etc., test case quality metrics are extensively used. Previous failing information of a test case is one of the existing quality metrics. This metric provides higher ranking to the test cases that failed in any of the previous releases. Higher rank of a test case suggests that the test case has higher probability to detect more faults in the current release, as well. However, in practice, the fault revealing test in the current release may not be exactly the same as the previous failed tests, instead, they might be similar, specially when new tests are added to the existing test suite or old tests are modified.

Therefore, I have proposed a similarity-based test quality metric that uses historical failure data. I have conducted a large empirical study (227 versions from five real-world java projects with real faults) that shows the proposed similarity-based metric is more effective in identifying the first fault of a system compared to the other traditional coverage and change-related metrics. I have also showed that combining this metric with existing metrics in a prediction model can result in a better

test quality metric.

6.1 Future Work

One potential future work is improving the studied similarity functions by abstracting the method sequence calls into a state model. To define the similarity, I have considered only the method name sequences from the previous execution traces that do not include specific data values. In the future, the method sequence calls from the execution traces could be merged into the state model based on the data values.

Another future direction is assessing the proposed similarity-based quality metric in terms of test case generation. Automated test case generation tools generate test cases based on different criteria such as branch coverage, statement coverage, mutation score etc [Fraser and Arcuri, 2011]. The proposed similarity-based metric could be used as an additional criterion for the automated test case generation tool, where the tool can generate tests with high coverage that are also similar to the previously failing tests.

Bibliography

P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008. ISBN 9781139468671.

J. Anderson, S. Salem, and H. Do. Improving the effectiveness of test suite through mining historical data. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 142–151. ACM, 2014.

A. Arcuri. 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*, pages 469–478. IEEE, 2010.

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

M. Cataldo, A. Mockus, J. Roberts, J. D. Herbsleb, et al. Software dependencies, work dependencies, and their impact on failures. *Software Engineering, IEEE Transactions on*, 35(6):864–878, 2009.

S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

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

defects4j. `http://homes.cs.washington.edu/~rjust/defects4j/`. [Online; last accessed 26-Oct-2015].

G. Dong and J. Pei. *Sequence Data Mining*. Advances in Database Systems. Springer US, 2007. ISBN 9780387699370. URL `https://books.google.ca/books?id=GESJmZpkePIC`.

Eclipse-AspectJ. The aspectj project. `https://eclipse.org/aspectj/`. [Online; last accessed 26-Oct-2015].

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

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.

M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

R. Feldt and A. Magazinius. Validity threats in empirical software engineering research-an initial survey. In *SEKE*, pages 374–379, 2010.

G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419. ACM, 2011.

G. Fraser and A. Arcuri. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):8, 2014.

GLM. `https://cran.r-project.org/web/packages/glm2/glm2.pdf`. [Online; last accessed 26-Oct-2015].

J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, (2):156–173, 1975.

K. J. Goulden. Effect sizes for research: A broad practical approach, 2006.

D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. ISBN 9780521585194. URL `https://books.google.ca/books?id=Ofw5w1yuD8kC`.

H. Hemmati and L. Briand. An industrial investigation of similarity measures for model-based test case selection. In *21st International Symposium on Software Reliability Engineering (ISSRE), 2010*, pages 141–150. IEEE, 2010.

H. Hemmati, A. Arcuri, and L. Briand. Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(1):6, 2013.

Y.-C. Huang, K.-L. Peng, and C.-Y. Huang. A history-based cost-cognizant test case prioritization technique in regression testing. *Journal of Systems and Software*, 85 (3):626–637, 2012.

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

R. Just, D. Jalali, and M. D. Ernst. 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*, pages 437–440. ACM, 2014.

A. Khalilian, M. A. Azgomi, and Y. Fazlalizadeh. An improved method for test case prioritization by incorporating historical test case data. *Science of Computer Programming*, 78(1):93–116, 2012.

J.-M. Kim and A. Porter. 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*, pages 119–129. IEEE, 2002.

S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller. Predicting faults from

cached history. In *Proceedings of the 29th International Conference on Software Engineering (ICSE), 2007*, pages 489–498. IEEE Computer Society, 2007.

H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

A. Mehta. Move fast & don't break things - google slides. `https://docs.google.com/presentation/d/15gNk21rjer3xo-b1ZqyQVGebOp_aPvHU3YH7YnOMxtE/edit#slide=id.g3f5c82004_8611`. [Online; last accessed 21-Dec-2015].

R. Moser, W. Pedrycz, and G. Succi. Analysis of the reliability of a subset of change metrics for defect prediction. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 309–311. ACM, 2008.

N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering (ICSE), 2005*, pages 284–292. IEEE, 2005.

N. Nagappan, L. Williams, M. Vouk, and J. Osborne. Using in-process testing metrics to estimate post-release field quality. In *18th International Symposium on Software Reliability (ISSRE), 2007*, pages 209–214. IEEE, 2007.

M. P. *Foundations of Software Testing*. Pearson Education, 2008. ISBN 9788131707951. URL `https://books.google.ca/books?id=yU-rTcurys8C`.

H. Park, H. Ryu, and J. Baik. 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*, pages 39–46. IEEE, 2008.

M. Pezzè and M. Young. *Software testing and analysis: process, principles, and techniques*. Wiley, 2008. ISBN 9780471455936.

Predict. `https://stat.ethz.ch/R-manual/R-devel/library/stats/html/predict.glm.html`. [Online; last accessed 26-Oct-2015].

R-Project. The R Project for Statistical Computing. `http://www.r-project.org/`. [Online; last accessed 26-Oct-2015].

D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič. Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55 (8):1397–1418, 2013.

G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 179–188. IEEE, 1999.

R:Ranks. R:Sample Ranks. `https://stat.ethz.ch/R-manual/R-devel/library/base/html/rank.html`. [Online; last accessed 26-Oct-2015].

K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification*, pages 419–423. Springer, 2006.

E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan. 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*, pages 300–310. ACM, 2011.

E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 62. ACM, 2012.

A. Vargha and H. D. Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

Q. Xie and A. M. Memon. Studying the characteristics of a "good" gui test suite. In *17th International Symposium on Software Reliability Engineering (ISSRE), 2006*, pages 159–168. IEEE, 2006.

S. Yoo, R. Nilsson, and M. Harman. 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/FSE11), Szeged, Hungary*, 2011.

H. Zhu, P. A. Hall, and J. H. May. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, 29(4):366–427, 1997.

T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Inter-*

national Workshop on Predictor Models in Software Engineering, PROMISE'07: ICSE Workshops 2007, pages 9–9. IEEE, 2007.