Master Thesis

September 11, 2017

Smart Prioritization for Tests in Test Suite Generation

Analysis of multiple ranking methods for effectiveness

Timofey V. Titov

(14-737-415)

supervised by

Prof. Dr. Harald C. Gall

Dr. Sebastiano Panichella

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Master Thesis

Smart Prioritization for Tests in Test Suite Generation

Analysis of multiple ranking methods for effectiveness

Timofey V. Titov

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Master Thesis

Author: Timofey V. Titov, timofeyvyacheslavovich.titov@uzh.ch

Project period: 10.04.2017 - 10.10.2017

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

I would like to thank Prof. Harald Gall for giving me the opportunity to write the thesis at the s.e.a.l. lab. Sebastiano Panichella's essential guidance and suggestions for this project are greatly appreciated. Special thanks go to Haidar Osman for providing predictions of his fault prediction model for the Defects4J dataset used in this thesis.

Abstract

Software and regression testing are to this day an integral part of software engineering. Companies spend billions of USD every year on Quality Assurance (QA) worldwide. A promising new approach to testing that automates even the coding part has emerged. It is receiving more attention from organizations in the industry, especially when rapid releases are becoming the norm which makes traditional regression testing impractical.

Automated Software Testing (AST) has many benefits such as reduced human labor and improved test coverage. However, one of the major limitations is that it takes minutes to generate tests for just one class. This is a problem considering that developers already have to wait hours to get feedback on their commits from a Continuous Integration (CI) server. This thesis tries to address the question whether smart ordering of classes under test makes discovery of unchecked exceptions faster by automatically generated tests.

The project repositories Joda-Time, Commons Math and Commons Lang from the Defects4J database of bugs are used for experiments, which are conducted with the test generation tools EvoSuite and Randoop. The main contributions of the thesis are: simulation involving half a dozen ranking methods and a ranking technique based on a novel code coverage prediction model. The primary performance metric is based on Area under the Curve that takes into account ideal ordering, as well as random ordering. The most sophisticated ranking method involving a combination of bug density and coverage prediction scores has positive results in all cases except for one involving the Joda-Time repository and Randoop test generation tool. However, the results are not statistically significant primarily due to high standard deviation.

Zusammenfassung

Software- und Regressionstests sind sogar heutzutage ein integraler Bestandteil von Softwareentwicklung. Unternehmen geben Milliarden von US-Dollar an Qualitätssicherung aus. Ein vielversprechendes Verfahren zum Testen hat entstanden, dass sogar das Programmieren automatisiert. Es hat mehr Aufmerksamkeit von Unternehmen in der Industrie erregt, besonders wenn das Rapid-Release-Prinzip etwas ganz Normales geworden ist. Das macht die traditionellen Regressionstests unpraktisch.

Automatische Softwaretests haben viele Vorteile wie z.B. reduzierte Arbeitskräfte und verbesserte Testabdeckung. Ein großer Nachteil ist jedoch Test-Erstellung die ein paar Minuten dauert für eine einzige Klasse. Das ist ein Problem, weil Softwareentwickler schon mehrere Stunden warten müssen um Ergebnisse über ihre Commits von einem Continuous-Integration-Server zu erhalten. Diese Arbeit beschäftigt sich mit der Frage von intelligenter Ordnung von Klassen die getestet werden und schneller Entdeckung von Ausnahmen die nicht als Checked-Exceptions gelten mit Hilfe von automatisch ertstellten Tests.

Die Projektarchive Joda-Time, Commons Math und Commons Lang von der Defects4J Dateibank mit Programmfehlern werden für Experimente genutzt. Die Experimente sind mit Testerstellungswerkzeugen EvoSuite und Randoop durchgeführt. Eine Simulation die sechs Rankingmethoden umfasst und ein Rankingverfahren das auf einem neuem Testabdeckungmodell basiert sind die Hauptbeiträge dieser Arbeit. Die Hauptleistungsmetrik ist auf Area under the Curve basiert und berücksichtigt ideale und zufällige Ordnung. Das hoch entwickelte Rankingverfahren ist eine Kombination von Programmfehlerdichte- und Testabdeckungvorhersagewerten. Es hat positive Ergebnisse in allen Fällen außer dem Projektarchiv Joda-Time und Testerstellungswerkzeug Randoop. Die Ergebnisse sind jedoch nicht statistisch signifikant wegen hoher Standardabweichung.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Motivation

Computers and electronic systems have been an integral part of our lives. Now the number of devices we come in touch has increased dramatically: smartphones, tablets, wearables and Internet of Things (IoT) devices. It is expected that by 2020 there will be 24 billion interconnected devices [GBMP13]. Another recent trend is the rise of so-called Big Data [JW14] applications, where new types of data are collected and analyzed to gain insights or enhance existing systems with predictive models. From a software engineering point of view this means that adaptation of existing software systems is just as true today as it was in the past [Leh80]. Software needs to be maintained, and with change new defects can be expected in addition to undiscovered ones.

Software defects or simply bugs is something that we encounter frequently as end-users in a computerized system or developers in a software library. These range from minor problems that affect user experience to critical ones where a system becomes inaccessible, or worse does the wrong thing and inflicts damage. Some of the bugs can be nondeterministic and hard to fix, while others can be security threats, but will not be fixed by the vendor [IC]. According to a study by Britton et al. half of the programming time by a developer is spent on debugging which has a yearly cost of 312 Billion dollars [BJC+13].

Software defects are dealt with in a number of ways including but not limited to: Quality Assurance (QA) testing, regression test suites, code review and static code analysis tools. According to an industrial survey of executives organized by Capgemini, Sogeti and HPE [Cap] the spend on QA and testing has been primarily rising over the past years to 31% of total IT budget in 2016. We can make an observation that this is smaller than the actual fraction spent on debugging according to Britton et al. [BJC+13] This is something that is widely present in the industry: bugs are often fixed in software after it is already "shipped". Interestingly enough, 41% of the respondents said that their organizations depend on manual testing.

A promising new approach to QA is automated software testing (AST) where not just the execution of the test is automated, but the generation (i.e. coding) as well. While the current research prototypes have many limitations, the vision is 100% automated testing [Ber07]. However, there is currently a lot of inertia and skepticism: in a recent survey from 2012 80% of respondents disagreed with the vision that automated testing will completely replace manual testing [RMPM12].

The main benefit of AST is that it saves human labor. This impacts the cost of testing, since machine time is cheaper than professional labor. According to a study by Rafi et al. [RMPM12] reliability and reusability of tests are listed as benefits as well. Since this is a new piece of technology it is natural to be skeptical about how well it does at testing software code. One should also keep in mind that 3 minutes of CPU time per class to generate a test suite is not a negligible resource cost considering that many repositories contain thousands of classes.

1.2 Research Goal

As mentioned previously automated software testing tools such as EvoSuite and Randoop is a promising direction for quality assurance of software. Even though these tools offer automation for code generation, they have their own limitations. Campos et al. [CAFA14] have shown what it is like for organizations to start using Automated Software Testing (AST) in a Continuous Integration (CI) server as part of a development process. In the study by Shamshiri et al. [SJR+15] it was shown that AST can detect real bugs successfully. This thesis can be thought of as a continuation of both studies. More sophisticated ranking methods for classes under test are explored. The ranking methods are based on LOC, predicted number of bugs, predicted bug density, predicted AST coverage and a combination of the previous two.

RQ1. Is the most sophisticated ranking method (combination of coverage prediction and bug density) effective at detecting unchecked exceptions?

RQ2. Does the most sophisticated ranking method remain effective at a smaller test generation budget?

RQ3. Is the human inspection effort for detected bugs smaller than for undetected ones?

Here, effectiveness means that the ranking method puts classes with detectable unchecked exceptions to the front. It is measured with an Area under the Curve fraction (AUC-f) metric that tells how close we are to the ideal ranking. The baseline is a random ordering. The Defects4J database of bugs used here provides a unique environment for research that is in some ways very close to regression testing in real-life. It should also be noted that the coverage prediction model has not been implemented in previous research and is another contribution of this thesis. While the research questions are specific, different metrics are collected for the other ranking methods to provide a context for discussion. For the third RQ the average LOC of detected buggy classes is measured, because it serves as a proxy for human inspection effort.

1.3 Structure of the Thesis

The second chapter consists of related work. The covered literature spans benefits and limitations of Automated Software Testing (AST) and factors that affect the use of automated testing in organizations. This serves as motivation and possible areas of application for AST. There is an overview of research on how automated tests detect faults. Finally, related papers regarding regression and test optimization are discussed. Since automated test generation is quite different from traditional regression testing more emphasis is put on the parallels with the work by Campos et al. They introduced the paradigm of Continuous Test Generation (CTG) involving EvoSuite in a Continuous Integration server environment.

The third chapter is about methodology and settings. The Defects4J dataset is described. Its strengths are mentioned, but also the experimental setup that it dictates. It consists only of buggy classes, and this is what is used for ranking throughout the thesis. The ranking methods for test prioritization are introduced here. In addition to that the main performance metric Area under the Curve fraction (AUC-f) is defined. This is followed by other experimental details such as test generation parameters used in the simulation.

In the fourth chapter the coverage prediction model is described in detail. It is another major contribution of this thesis. It requires a separate training dataset. The prediction model relies on package-level, Chidamber and Kemerer (CK), Object-Oriented (OO) and Java reserved keyword features derived from source code. The efforts put into tuning and selecting a machine learning algorithm are described.

The main results are presented in the fifth chapter. The ranking methods are evaluated primarily with the AUC-f metric. Performance of the most sophisticated method COVERAGE_DENSITY_COMBO is examined in detail at different test generation budgets. The third research question regarding human inspection effort of bugs is addressed. This is followed by a discussion of ranking method performance in terms of branch coverage. In the last chapters there is a discussion of results, threats to validity and conclusion where research questions are addressed again together with a future outlook.

Chapter 2

Background and Related Work

2.1 Background Information

2.1.1 Regression Testing

Regression testing is a form of testing where it is ensured that the system has the core functionality it is supposed to have. Such test suites are executed regularly when source code is changed to verify that no new bugs are introduced. Since each test case is meant for a specific behavior, there is many of them. This has led to the question of prioritization of regression testing early on [RUCH01]. Historically, such test cases were performed manually. Over time more test automation is introduced including automated GUI testing [AFR15]. Regression testing is still very relevant today in modern software development methodologies such as Extreme Programming (XP). Automated Software Testing tools like EvoSuite is the next logical step, because even writing of the test case is automated.

2.1.2 Automated Software Testing (AST)

In this thesis automated testing means testing through automated unit test generation unless otherwise specified. This process is meant to assist software developers with the task of testing. There exist three main approaches for AST: *random testing*, *Search-based Software Testing (SBST)* and *Dynamic Symbolic Execution (DSE)*. In this thesis the tools EvoSuite [FA11] and Randoop [PE07] from the first two categories are used. The majority of the tools even those for DSE involve some form of randomized algorithm for exploration of the code. Another common characteristic among all approaches is that the test suite generation step is the most expensive one. Both of these reasons serve as motivation for the model introduced in this thesis which predicts the coverage achieved by EvoSuite or Randoop beforehand.

2.1.3 Test generation tools

Randoop and EvoSuite are the two test generation tools used in this project. Randoop [PE07] belongs roughly to the category of *random testing*. This is when a random input is provided to a class by building up complex inputs from Java primitives. Randoop is smarter than just random testing, and the authors refer to it as *feedback-directed random testing*. The execution feedback of such random inputs is used "to avoid generating redundant and illegal inputs" [PE07]. It even contains built-in contracts (i.e. invariants) that need to be always satisfied: two objects that are equal according to `object.equals()` should have equal Java `hashCode()`.

EvoSuite [FA11] is based on Search-based Software Testing. There are also randomly generated inputs, but they are evolved with a Genetic Algorithm guided by a fitness function that keeps track of code coverage and number of exceptions during testing. What makes EvoSuite unique is that it has included numerous automated oracles. This is done through code transformations. For example, before a division by a variable x the following code would be inserted: `if(x == 0) throw new ArithmeticException()`. Now, if there is an input that causes division by zero, then the fitness function will assign a higher score to that input, because an exception is triggered. Such sophisticated tricks and heuristics have allowed EvoSuite to perform better than other tools in recent automated unit test competitions [PM17].

2.1.4 Benefits and Limitations of Automated Testing

Rafi et al. [RMPM12] did a survey on automated testing literature by considering papers published during the time period of 1999-2011. The source papers focus on benefits and limitations, and are primarily of empirical nature (experiments and industrial cases). They also found that limitations are primarily described in experience papers.

ID	Title
B1	Improved product quality
B2	Test coverage
B3	Reduced testing time
B4	Reliability
B5	Increase in confidence
B6	Reusability of tests
B7	Less human effort
B8	Reduction in cost
B9	Increased fault detection

Table 2.1: AST benefits according to the survey by Rafi et al. [RMPM12].

We can see from the table 2.1 that many benefits come as a result of automation such as test coverage, reduced testing time, reliability, reusability of tests, less human effort and reduction in cost. This thesis is primarily concerned with further efficiency and coverage, as well as fault detection which corresponds to benefits B1, B2, B3, B8, B9.

From the list of limitations in table 2.2 we can see that one cannot neglect the effort required by software engineers in order to setup and maintain AST. The higher cost of engineering labor offsets the savings in the QA department. Still, even though tooling improves every time, it is not clear whether AST can completely replace manual testing. Certain testing tasks are very complex and require domain-specific knowledge. One can argue that the experiments of this thesis show that limitation L2 (failure to achieve expected goals) may not be relevant anymore.

The study [RMPM12] authors also did a practitioner survey to compare views with the academic perspective. "Overall, it is visible that the benefits of AST that were found in literature are strongly supported by the respondents, with at least half of them agreeing or fully agreeing to 8 out of 9 statements. Only for rank 9 (high fault detection) more than half of the respondents are either indifferent or disagree/strongly disagree" [RMPM12]. The practitioners also agreed with limitations presented here. For example, 45% of respondents found available tools to be a poor fit and 80% disagreed with the vision that manual testing would be replaced by automated testing.

ID	Title
L1	Automation can not replace manual testing
L2	Failure to achieve expected goals
L3	Difficulty in maintenance of test automation
L4	Process of test automation needs time to mature
L5	False expectations
L6	Inappropriate test automation strategy
L7	Lack of skilled people

Table 2.2: AST limitations according to the survey by Rafi et al. [RMPM12].

2.2 Automated Testing in Organizations

In this section the benefits and limitations, as well as other factors become apparent if we think about an organization deploying automated testing. This provides a context for a possible application of the ranking methods developed in this project.

Reasons for using automated testing

We should also consider the fact that organizations move to rapid release schedules. This is arguably another reason for companies to use AST. Rapid releases have many benefits such as faster time-to-market and software evolution. In the case study on software testing in rapid releases [MAK+15] Mäntylä et al. found that this resulted in a narrower test scope with a deeper focus, regression tests with the highest risk and testing of continuous nature. The landscape of software companies is competitive who employ methodologies such as agile development. This pressure may result in greater development of AST tools, because it addresses key challenges in this situation (with references to strengths from table 2.1):

- physical time constraints between deadlines (B1)
- narrow scope of tests (B2)
- scarcity of QA engineers and human resources (B7)

Factors that affect the use of AST

Garousi and Mäntylä [GM16] did a Multi-vocal Literature Review (MLR) on when and what to automate in testing which consisted of 52 grey literature and 26 academic sources. Many of the factors are relevant for the kind of automated testing that is covered in this thesis. Here, the factors identified by Garousi and Mäntylä, which form a test factor taxonomy, are presented.

SUT-related

The maturity of System under Test (SUT), planned development and constant change directly affect the amount of testing effort. "Automation fails when the current application has unstable design" was said in an interview by a test engineer from United Health Group [GM16]. This is ultimately an economic factor as well. While test engineers can manually mock out behavior of external third party components it is very difficult to do so completely automatically especially when multiple programming languages are involved.

Test-related

Under *test-related* factors the authors [GM16] list the need for regression testing. This applies to tests generated by tools like EvoSuite. Even though the cost of automatically generated tests may be low, it's not free. Certain parts of the system don't strictly require regression testing. Test reuse and repeatability is relevant here and coincides with limitation L3 [RMPM12] which has also been noted in other works [KRTS09].

Test-tool related

The tools themselves have limitations which need to be considered during review. Beyond effectiveness an organization needs to evaluate dependencies and infrastructure requirements. In most cases a combination of tools is necessary. For example, Evosuite and Randoop don't support GUI testing. At the same time it would be best for an organization not to depend on a single vendor for its testing needs: "Popular open-source tools are often good options as they have a low cost (e.g., only training, etc.), do not have the risk of a single tool vendor (such as increasing prices, sudden stopping of tool development, etc.), and large user bases which can be seen as the insurance of the tools' future sustainability and success" [GM16].

Human and organizational

Human and organizational factors also play a role. These include the skills level of testers and management priorities. Considering the complexity of the setup and maintenance in AST an organization would need the expertise of software engineers. Since AST is still relatively new in the industry, training costs for staff might be prohibitively high (compare with L7).

Cross-cutting

We have already seen that economic factors play a big role for adoption of AST. The other major factor that spans multiple factor categories is the development process. With the move towards rapid releases it appears that there is a need towards lightweight testing design with the help of tools like EvoSuite. Interestingly enough, Garousi and Mäntylä [GM16] point out that the Waterfall development model may not benefit much of automated testing since it is performed as a stand-alone phase.

2.3 Automated Tests for Finding Faults

2.3.1 Research performed on Defects4J

The closest work to this thesis is that of Shamshiri et al. [SJR+15] This is one of the first major works on detecting real faults as opposed to seeded faults with the help of automated tests. The experimental setup is almost identical: among the tools are EvoSuite and Randoop, the dataset is Defects4J and the test generation time is 3 minutes per class.

They found that automated tests detected 55.7% of the faults in Defects4J. They found that 16.2% of all faults and 36.7% of the non-found faults were not even covered during execution. This is a supporting reason for coverage as an important criterion in generated tests. According to the authors "EvoSuite and Randoop generated 3.4% and 8.3% non-compilable test suites on average" which had to be filtered out.

The experimental setup of this project is almost identical to that of Shamshiri et al. Thus, it is important to mention what they have done to ensure that exceptions triggered correspond to bug detection. They have manually investigated that the exception corresponds to the fault for a subset of data. However, for the rest of the data they mainly relied on a sanity check involving a test exception such that it is the "same exception or a similar assertion" [SJR+15].

2.3.2 Other research on bug detection

There are other works on detecting real faults with automated tests. A similar setup to that in Defects4J was applied to 25 real faults of a proprietary application in the work by Almasi et al. [AHF+17] Results are comparable to this thesis and the work by Shamshiri et al. [SJR+15]: bug detection of 56% for EvoSuite and 38% for Randoop. In the study they also do a practitioner survey to find out about barriers.

The survey revolved primarily around setting up the tool, executing test suites, resolving dependencies and readability of the tests. According to the taxonomy by Garousi et al. [GM16] this survey is concerned with SUT, test-tool and human factors. For the first three categories most developers agreed on the easiness of the process. This also addresses limitations *failure to achieve expected goals* (L2), *difficulty in maintenance of test automation* (L3) and *process of test automation needs time to mature* (L4) from the survey by Rafi et al. [RMPM12] For example, EvoSuite has support for Integrated Development Environment (IDE) like IntelliJ and Continuous Integration (CI) server such as Jenkins through Maven integration. This is a sign that automated tests become more promising for real-life deployments.

Another major work on discovering real faults with the help of automated tests is that by Fraser et al. [FA15]. It is performed on a much bigger scale with 100 open-source projects from SourceForge [FA12]. Another difference is that all classes are tested as opposed to a specially prepared set of classes like in Defects4J. Triggered exceptions are then sampled for manual verification that it is indeed a real fault. Among the 6376 out of 8844 classes EvoSuite found 32,594 exceptions. Violations of `assert` statements in code is a clear indication of a violation and EvoSuite found 477 of those.

Artho et al. [AM16] manually evaluated 208 test case failures from a tool called GRT [MAZ+15]. This tool has a hybrid approach between random testing and Dynamic Symbolic Execution, and it was run against ten popular open-source projects including Apache Math that is part of Defects4J. Their existing workflow already filters out false positives. However, among the 90 filtered test failures only 56 resulted in real issues. They found that many tests are invalid, because these tests don't take into account indirect specification and documentation. At the same time "about one third of the issues were addressed by updating the documentation but not the program code" [AM16]. This is one of the main reasons for false positives, and also shows how change of existing code is error-prone without expert knowledge which is a common situation in companies with legacy code. The authors give additional reason for machine-readable documentation such as the `@NonNull` annotation in Java 8. This study is also important to reflect upon the usefulness of test cases of AST. It appears that developers should consider such feedback during development as part of CI server report as envisioned in previous works on EvoSuite [FA15, CAFA14].

2.4 Test Optimization

2.4.1 Regression and acceptance test optimization

There have been many studies on prioritizing regression tests in order to reduce resource usage [RUCH99, WHLA97, KP02]. The growing nature of regression tests remains a problem today.

Even Google reports that it is prohibitively costly to execute all regression tests [GIP11]. Recently, Shi et al. [SYGM15] did a study on regression test suite optimization and measuring the effect on fault-detection capability based on mutants. It is the "first extensive study that compares testsuite reduction and regression test selection approaches individually, and also evaluates a combination of the two approaches". [SYGM15] Their best model provides as high as 45 percentage points reduction in tests with a loss of 5.93% of change-related fault (mutant) detection. On the other hand, in the study by Lu et al. [LLC+16] it was shown that traditional and time-aware test prioritization techniques become less effective, if one also considers test updates during software evolution. This is something not accounted for in most test prioritization experiments partially due to the use of mutants instead of real faults.

The study by Monden et al. [MHS+13] deals with how fault prediction models can help with software quality and reduction of costs. This is an interesting angle, because previously fault prediction studies have primarily focused on accuracy [FN99,HBB+12]. For this purpose they trained their model on 4 releases of a private software system, and simulated the effects of multiple test effort allocation strategies on acceptance tests. Their best model consisted of allocating effort according to "the number of expected faults in a module · log(module size)" [MHS+13]. They used a fault detection simulation called exponential Software Reliability Growth Model (SRGM) that has a constant fault detection rate at an arbitrary testing time. According to the simulation the test effort was "reduced by 25 percent while still detecting as many faults as were normally discovered in testing".

Wang et al. [WNT17] prioritize regression test cases by considering defect prediction model CLAMI [NK15] and static analysis tool FindBugs [HP04]. Their goal is to order test cases in such a way that faults are detected earlier. The motivation is that existing methods "do not take the likely distribution of faults in source code into consideration" [WNT17]. An example of one of their best ranking methods is JMT-BF: it extracts static method coverage information from existing test case, and combines it with bug finder information.

$$QMCoverage[t_i] = \sum_{j=1}^{m} cover(t_i, mc_j) \cdot mw_j$$

In the formula above a method-level quality-aware score is given to a test case t_i used for prioritization. $cover(t_i, mc_j)$ means that method mc_j is covered by test case t_i. mw_j is a weight computed for a method from a Class under Test based on FindBugs output. The results on 7 open-source projects indicate that their quality-aware test prioritization can improve existing coverage-based techniques for up to 15.0% and on average 7.6% for regression tests in terms of Average Percentage Fault Detected.

Coverage-based test prioritization [HPH+16,LLC+16] has been shown to have state-of-the-art performance for optimizing regression testing. In many experimental setups total code coverage of a project is even the main goal. Besides test prioritization approaches [GEM15, JZCT09, KNY+15] there is work on test selection [Bal98,SYGM15] and reduction [SGG+14,BMK04]. However, the setting of AST is quite different, because the generated tests are unit tests with fast execution time compared to integration tests. It should also be noted that test minimization is already built into tools like EvoSuite. The most expensive step in AST is test generation. In order to use real faults instead of mutants the Defects4J database of bugs is used. Its main strength is that it isolate bug-related changes and provides two versions for each data point: a fixed and a buggy version. The disadvantage is that there is a limited number of classes (357) that are marked like this. Thus, we can only rank analyzed classes (known to be faulty) which also makes it hard to compare with traditional experimental setups in regression testing research.

2.4.2 Continuous Test Generation (CTG)

The research by Campos et al. [CAFA14] has similar goals as this thesis for optimizing the use of resources for automatic test generation and discovering unchecked exceptions. Their ranking characteristic is number of Java branches in a compiled class. This is a slightly better metric than LOC for the purposes of measuring complexity, but not necessarily for the purposes of bug detection, because source code is ultimately what programmers see and use.

However, their work has also big differences compared to this thesis. Their main goal is in optimizing generation for a project as a whole on a Continuous Integration (CI) server. This addresses limitation L4 as well as expectation voiced by one of the developers regarding CI server Jenkins support in the study by Almasi et al. [AHF+17] Their other optimizations include prioritizing classes according to how much they were changed in previous commits, as well as reusing seeds for the Genetic Algorithm (GA) in EvoSuite from previous runs.

According to their smart budget allocation and ordering the branch coverage is improved by up to 58% while thrown undeclared exceptions go up by 69%. Although one must admit that you gain a lot of improvement by focusing on classes with the most changes as part of a commit. They also motivate the need for a coverage prediction which was for the first time implemented in this thesis:

> The budget allocation optimization has to be done before generating any test case for any CUT, but the values t(c, z) = y are only obtained after executing the testing tool and the test cases are run. There is hence the need to obtain an estimate function t' , as t cannot be used.

Here, t(c,z) is a performance function for a tool like EvoSuite with parameters (i.e. characteristics) for a class (c) and budget (z). The need for this becomes apparent if we consider differences to traditional regression testing. One of the reasons is that automatically generated tests cannot be currently automatically extended as software evolves. This means that tests need to be regenerated when software changes. However, existing techniques rely on static and dynamic code coverage analysis for existing tests. In the case of AST the most expensive step is test generation, i.e. creation, and code coverage information is not available.

Chapter 3

Methodology and Setup

3.1 Defects4J Database of Bugs for Research

It is important to go over characteristics of the Defects4J [JJE14] database of bugs, because it dictates a certain type of experimental setup. The Defects4J database used in this thesis contains faults of real-life open-source projects. This is its main advantage compared to other datasets for testing research which rely on hand-seeded faults or mutants. For this thesis we consider 4 open-source projects from Defects4J that are based on the *git* version control system: Closure Compiler, Apache Commons Math, Joda Time and Apache Commons Lang. Each of these projects contains tens of thousands of LOC. The bugs considered as part of Defects4J follow a specific set of criteria [JJE14]:

- it is related to the source code (and not the build script for example)
- it is reproducible with the existing test suite
- it is isolated (the fix doesn't contain unrelated changes)

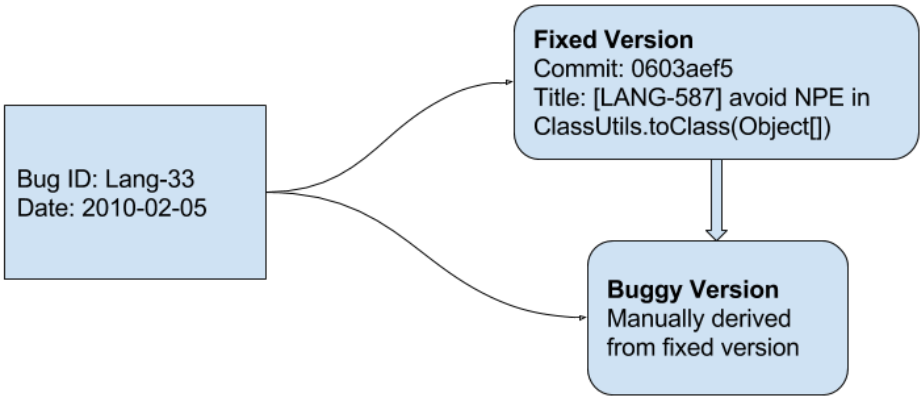

Figure 3.1: Conceptual diagram for the bug Lang-33 from Defects4J. There is always a fixed and a buggy version available. The latter is manually derived by the authors from the fix by undoing it.

Not only is it a database of bugs, but it has built-in infrastructure that facilitates interaction such as checking out code (regardless of the type of VCS), building the source code etc. Its infrastructure has been leveraged in previous research as well [PM17]. The Defects4J parameters are used whenever possible including test generation tool configuration.

3.2 Overview of Approach

I am simulating regression testing performed by tools such as EvoSuite and Randoop instead of manually written tests. The experiments are repeated 3 times due to long running time (6 server days in total). Essentially, each data point that we deal with here is a class known to be buggy. Dependence on Defects4J becomes apparent. I generate tests for the fixed version of a program and run the test on the buggy version. If the test throws an unchecked exception, then this is considered a success, because it must have caught a bug. It is now also clear why the bugs in the Defects4J dataset are isolated: we don't want the test to fail simply because version V_{bug} has different source code functionality than V_{fix}.

Shamshiri et al. [SJR+15] test whether automatically generated tests are capable of detecting bugs in Defects4J. In this thesis the direction is very similar but deeper, since effectiveness of testing is evaluated. This is done by considering different CPU time allocations, as well as ranking methods for the order in which classes are tested. The reader may be surprised that since Defects4J contains only buggy classes that only those classes are ranked. This is still useful, because there is a continuum of error-proneness among classes. Moreover, what is important in Automated Software Testing is feedback in the form of bug detection and unchecked exceptions. This is captured in this experimental setup, because it is measured how many of faulty classes are detected as bugs and whether they are ranked higher than the rest by the ranking method. This experimental setup is partially dictated by the Defects4J dataset and also due to the computational cost of generating tests.

3.3 Ranking Methods

3.3.1 Lines of Code (LOC)

Ranking by LOC is meant to capture complexity of a class. Here, LOC of the class under test is calculated beforehand. This is a simple and easy to interpret measure of complexity. It has been shown previously that there is a strong association with cyclomatic complexity [She88]. It is expected that under limited resources for test generation classes with smaller LOC will yield higher coverage. As a result of that there is a higher chance of bug detection. This metric has also other interesting properties. In studies on bug prediction it is used as a proxy for human inspection effort [MMT+10, PAP+16]. Here, its primary role is that of a complexity metric for the size of the search space for automatic test generation. This is the only method that sorts classes in ascending order according to the metric.

3.3.2 Coverage prediction

Another contribution of the thesis is a prediction model for coverage by EvoSuite. In a sense this is taking the idea of a complexity measure to the extreme. Test generation is expensive and we would like to know ahead of time what coverage will be achieved by the generated test suite. This idea was first proposed in the paper by Campos et al. [CAFA14] The model is described in detail in the next chapter.

3.3.3 Error proneness

The error proneness ranking methods include predictions for a class of interest: number of bugs and bug density, where bug density is predicted number of bugs normalized by LOC. This is a model by Haidar Osman [OGNL16] who also kindly provided predictions for the Defects4J repositories used in this thesis. The model uses the Random Forest ML algorithm to make predictions for each Defects4J faulty class based on recent history. An important feature of the model is that it goes beyond Defects4J bugs and considers other types of bugs for learning by analyzing commit messages. The two types of prediction scores are referred to as BUG and BUG_DENSITY throughout this write-up.

3.3.4 Coverage prediction and error proneness combined

Here, the scores of two prediction models are used to create a combination: coverage and bug density. First, both sets of predictions are scaled to the [0,1] range. This is necessary, because bug density values have tiny magnitudes. Then the harmonic mean is taken which creates the combined value. It "smooths out" the final score if one of the components is still extremely different.

$$CDC = \frac{2 \cdot C \cdot BD}{C + BD}$$

In this equation C and BD stand for coverage prediction and bug density prediction, respectively. This ranking method is known as COVERAGE_DENSITY_COMBO (CDC) in the plots and tables below.

3.3.5 Covered branches prediction

This is also a derivative of the coverage prediction method. It is coverage prediction score weighted by the total number of Java bytecode branches of a Class under Test (CUT).

$$CB = C \cdot B$$

The method is referred to as COVERAGE_BRANCH (CB) in the graphs and tables to follow. The coverage prediction score (C) is multiplied by total number of branches (B). This weighting pursues a different strategy, because it considers the size of the class. A larger class with a small percentage of covered branches will be ranked higher as long as the absolute number of predicted covered branches is high.

3.4 Evaluation and Metrics

3.4.1 Bug detection

The main criterion for success in this project is whether an automatically generated test detects a bug, and whether such an instance is ranked higher than other classes. Bug detection is implemented in the simplest possible way: Defects4J considers a test to fail if it triggers one or more exceptions on the buggy version. This raises the question of validity which overlaps with the problem faced by the related study of Shamshiri et al. discussed in related work section. In short, since the experimental setup is almost identical to the one by Shamshiri et al. [SJR+15] their checks support the approach here as well. This supports the use of the term "bug detection" throughout the thesis.

3.4.2 AUC-f

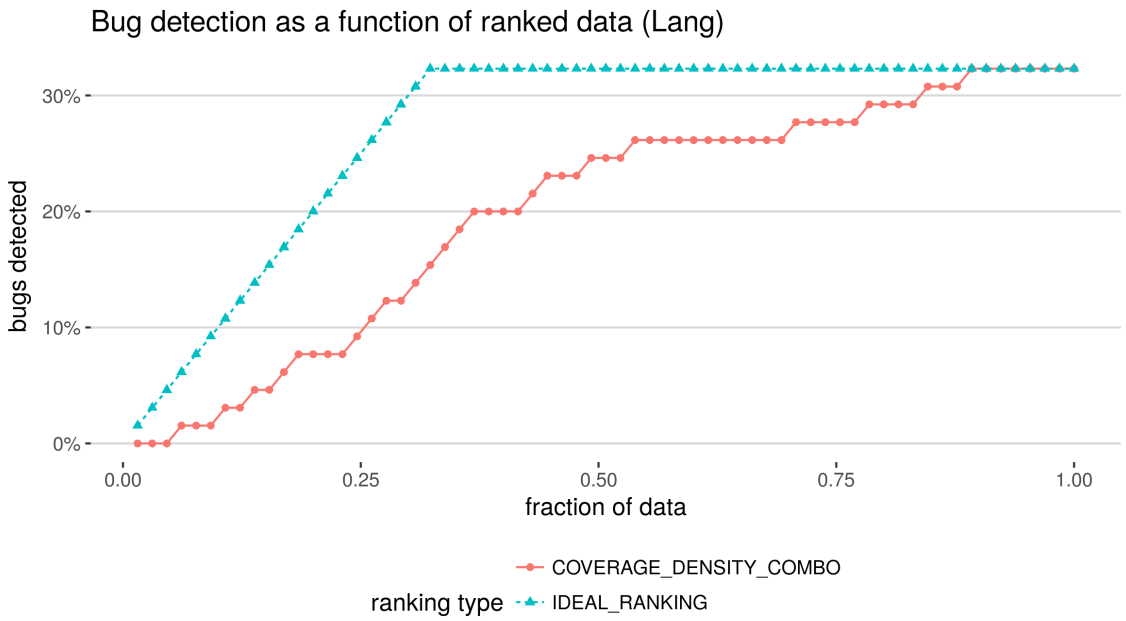

Figure 3.2: AUC plot for bug detection of the Lang project and EvoSuite testing tool. The best possible ranking is shown on top. The baseline of random testing is a diagonal connecting lower left and upper right corners which is equivalent to constant bug detection.

The diagram shown in Fig. 3.2 is the main visualization used to analyze performance. The idealized baseline is the lower triangle with a diagonal connecting lower left and upper right corners. It shows that if a class is tested in random order, then the success rate is the same, i.e. it is equal to the bug detection rate achieved in the end. An ideal ranking puts all the "detectable" buggy classes in the beginning and the curve forms a trapezoid. The ranking method considered here covers an area greater than the baseline.

The inspiration behind using AUC as a metric is the well-known ROC curve [HM82]. The curves considered here don't have such deep meaning, but the principle is the same: we want to cover higher amount of area. How good can the ideal ranking get? If the total bug detection rate is 100%, then the ideal and baseline curves coincide. However, the total bug detection rate usually doesn't go beyond 50%. In our case, the trapezoid covers a higher amount of area, because it has a steeper side. To take into account improvement over the baseline and distance from the ideal the following metric is proposed:

$$\frac{AUC(ranking) - AUC(baseline)}{AUC(ideal) - AUC(baseline)}$$

This is referred to as AUC-f and it is a fraction with an upper bound of 1 instead of an area. A value greater than 0 shows improvement over the baseline. Larger values mean closeness to the ideal which is preferable. The same metric can be used in the setting of branch coverage. In that

case the y-axis is for branch coverage, but the principle and calculations are the same.

3.5 Experimental Setup

3.5.1 Budget (CPU time allocation)

Generally speaking only a few minutes are given per class to test generation tools [SJR+15, FA15, PKT15]. To make this work comparable with that of Shamshiri et al. [SJR+15] I also use 3 minutes per class. This is still a generous allocation of resources. Since I am testing efficiency with the help of smart ordering, I am also considering other budgets: 45, 90, 180 (in seconds).

3.5.2 Data

Project	Bugs	KLOC	Dev. years
Joda-Time	27	28	11
A. Commons Math	106	85	11
A. Commons Lang	65	22	12
Google Closure	133	90	5

Table 3.1: Characteristics of the Defects4J repositories according to the authors [JJE14].

For this thesis I use 4 out of 6 repositories from the Defects4J database. Even though JChart is in the database it doesn't appear to be git-based which is a requirement for the error proneness method. Mockito is not used, because it had an open issue [gof]. Even though initially I used Closure, it is later dropped due to abnormal bug detection of a few percent.

3.5.3 Hardware

Name	X64-15GB
Architecture	x86-64
Number of cores	6
Memory	15GB
Disk type	SSD

Table 3.2: Hardware specifications for Scaleway server [Sca] used for experiments.

3.5.4 Implementation details

Just like other works [RJGV16, Pra13] on automated testing I rely on multi-core functionality of CPUs to speed up experiments. Even though the tasks here are trivially parallelizable there is extra complexity involved with execution and data collection. I had to make sure that tasks are

launched at multiple workers simultaneously which is a form of parallelization. Another implementation optimization is to generate tests with the same budget first before any ranking. This data is then used for ranking in different ways. In a real-life scenario one would rank classes first before generating tests.

Chapter 4

Coverage Prediction Model

4.1 Overview

Automatic test generation takes minutes for every class. It would be nice to know the coverage before spending computational resources. The tools Randoop and EvoSuite are themselves non-deterministic, because the test generation task is challenging. A prediction model for coverage is built with the help of machine learning, because there are no tests before the test generation phase. First, additional training dataset is constructed. Using this raw data tests are generated using EvoSuite and Randoop. Many features from source code are built that include package-level, Chidamber and Kemerer (CK), Object-Oriented (OO) and Java reserved keyword features. Finally, three different machine learning algorithms are tuned using cross-validation. The final model is based on Support Vector Regression and achieves a MAE of 0.216 and 0.0880 during cross-validation on EvoSuite and Randoop data. It is actually two separate models trained for EvoSuite and Randoop specifically.

4.2 Data for Prediction Model

To ensure integrity of the experiment a different dataset is used to train and build the coverage prediction model. The data consists of the following open-source projects: Apache Cassandra, Apache Ivy, Google Guava and Google Dagger. Apache projects are quite popular in software evolution and maintenance literature. [BCDP+13, ZNG+09] While Guava can be thought of to be similar to Apache Commons in Defects4J, the other projects are from different domains: Cassandra (distributed database), Ivy (build tool) and Dagger (dependency injector). The source files, classes and compile dependencies had to be manually collected from Maven Central [Mav].

	Guava	**Cassandra**	**Dagger**	**Ivy**
LOC	78525	220573	848	50430
Java Files	538	1474	43	474

Table 4.1: Summary of data characteristics used to build the coverage model. The statistics are computed using *cloc* [Dan].

These are raw data sources. Each class from these data sources is tested with EvoSuite and Randoop resulting in two separate datasets for further machine learning. The branch coverage is

computed by EvoSuite itself, while for Randoop I rely on a script from jdoop [jdo]. During this step classes with incompilable test cases are completely removed from the model. Target values of 0 and 1 of branch coverage are considered extreme, and removed from training data. After doing these filtering steps only about one fifth of observations remain.

4.3 Model Building

The features used for the model are meant to primarily capture code complexity. The first set of features comes from JDepend [Cla] and captures information about the outer context layer of the class under test. I also compute the well-established Chidamber and Kemerer (CK) [CK94] and Object Oriented (OO) metrics such as depth of inheritance tree (DIT) and number of static invocations (NOSI). This set of metrics is provided by a tool written by M. Aniche [Ani]. Finally, I go even more fine-grained and include counts for all 52 Java reserved keywords of the source code such as `synchronized`, `import` and `instanceof`. The last feature is CPU time allocation (budget) for test generation which is encoded as a categorical variable due to limited number of budgets considered here: 45s, 90s, and 180s.

4.3.1 Package-level features

Name	Description
TotalClasses	The number of concrete and abstract classes (and interfaces) in the package.
Ca	The number of other packages that depend upon classes within the package.
Ce	The number of other packages that the classes in the package depend upon.
A	The ratio of the number of abstract classes (and interfaces) in the analyzed package.
I	The ratio of efferent coupling (Ce) to total coupling (Ce + Ca) such that I = Ce / (Ce + Ca).
D	The perpendicular distance of a package from the idealized line A + I = 1.

Table 4.2: Summary of package-level features from the JDepend manual [Cla].

Originally, these features were meant to be indicators of package quality. For example, total number of classes is a measure of extensibility of that package. The features Ca and Ce can be thought of the package's responsibility and independence, respectively. Both are indicators of complexity for the purpose of coverage prediction. Another interesting combination feature is distance from the main sequence (D). It captures closeness to an arguably optimal package characteristic when the package is:

- abstract and stable (A=1, I=0) or
- concrete and unstable (A=0, I=1)

4.3.2 CK+OO features

This set of features contains recognizable Chidamber and Kemerer (CK) metrics such as WMC, DIT, NOC, CBO, RFC and LCOM. It is important to note that the ck tool [Ani] calculates these from source code directly using a parser. It is not clear from documentation which exact version of the many variants of LCOM is computed. In addition to these there's other OO features that are not so standardized such as number of static fields (NOSF).

Name	Description
CBO (Coupling between objects)	Counts the number of dependencies a class has.
DIT (Depth Inheritance Tree):	Counts the number of ancestors a class has.
NOC (Number of Children)	Counts the number of children a class has.
NOF (Number of fields)	Counts the number of fields in a class regardless of modifiers.
NOPF (Number of public fields)	Counts only the public fields.
NOSF (Number of static fields)	Counts only the static fields.
NOM (Number of methods)	Counts the number of methods regardless of modifiers.
NOPM (Number of public methods)	Counts only the public methods.
NOSM (Number of static methods)	Counts only the static methods.
NOSI (Number of static invocations)	Counts the number of invocations to static methods.
RFC (Response for a Class)	Counts the number of unique method invocations in a class.
WMC (Weight Method Class)	It counts the number of branch instructions in a class.
LOC (Lines of code)	It counts the lines of count ignoring empty lines.
LCOM (Lack of Cohesion of Methods)	Measures how methods access disjoint sets of instance variables.

Table 4.3: Summary of CK+OO features taken from the ck tool description [Ani].

4.3.3 Java reserved keyword features

```
abstract, assert, boolean, break, byte, case,
catch, char, class, const, continue,
default, do, double, else, extends,
false, final, finally, float, for,
goto, if, implements, import, instanceof,
int, interface, long, native, new,
null, package, private, protected, public,
return, short, static, strictfp, super,
switch, synchronized, this, throw, throws,
transient, true, try, void, volatile,
while
```

Listing 4.1: List of Java reserved keywords in alphabetical order used as features.

Keywords as features have long been used in IR [SC12]. However, to our knowledge Java keywords have not been used as complexity features. One possible reason for this is that these features are too fine-grained and don't allow for complexity thresholds like with CK metrics [BEEGR00]. There is definitely some overlap with other features from this model by considering counts of keywords: `abstract`, `class`, `static` etc. Intuitively, it is easy to see that for example `synchronized`, `import` and `switch` are indicators of code complexity.

4.3.4 Feature transformation

In order to bring magnitudes of feature values to comparable sizes they are first log-transformed and then normalized using a z-score (standard Gaussian distribution).

4.4 Model Training

For the prediction model multiple algorithms from Python's ScikitLearn library [Sci] are considered: Huber regression, Support Vector Regression and Multi-layer Perceptron. They are all constitute different families of algorithms, and the last two are non-linear. The algorithm selection procedure begins with a grid search over a wide range of values. The tuning is performed using the training data in table 4.1. For each set of hyper-parameters 3-fold cross-validation is done, the performance on the validation folds averaged and used for selection of best parameters. This way the learned model is regularized and should have comparable performance on new data such as that of Defects4J. The performance is measured in terms of Mean Absolute Error (MAE) which is easy to interpret, because it is in the same unit as the target variable (branch coverage fraction):

$$MAE = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n}$$

where y is predicted values and x are observed values.

Testing tool	Huber Regr.	Support Vector Regr.	Multi-layer Perceptron
"evosuite"	0.255	0.216	0.242
"randoop"	0.172	0.0880	0.139

Table 4.4: Best cross-validation MAE scores on training data from various machine learning algorithms. The algorithms perform consistently on EvoSuite and Randoop data with Support Vector Regression being the best.

Huber Regression [HR75] is a robust linear regression model. It is more tolerant to data that contains outliers. In case of points it detects as outliers it applies only linear loss to such observations which softens the impact on the overall fit. The parameter α for the model is explored over a wide range: 2 to the power of `linspace(-30, 20, num=15)`, where the powers are linearly spaced between -30 and 20 with 15 points in total. The best α values are 7420 and 624.1 for EvoSuite and Randoop training data.

The Support Vector Regression implementation in Python's Scikit-learn library is based on `libsvm` [CL11]. Here, it is used with a radial basis function kernel. It learns non-linear patterns in data by forming hyper-dimensional vectors from data and evaluating how similar new observations are to those from training. The main regularization parameter *C* is checked in the range of: 2 to the power of `linspace(-30, 20, num=15)`. Just like with Huber regression the powers are linearly spaced between -30 and 20 with 15 points in total. In addition to that the parameter ϵ is tweaked, which controls "the size within which no penalty is associated in the training loss function with points predicted within a distance epsilon from the actual value" [skl]. The values considered here are: 0.025, 0.05, 0.1, 0.2, and 0.4. The best hyper-parameters are: $C = 4.416$ and $\epsilon = 0.025$ for EvoSuite and Randoop data.

Finally, the last algorithm considered is a Multi-layer Perceptron. Neural networks come in different configurations, but generally have good performance. [NHC13] Another reason for using neural networks is that they are known for finding non-linear interactions between features which is desirable for this kind of data. A grid-search over a set of different hyper-parameters is performed:

- *alpha* values in the range of 2 to the power of `linspace(-30, 20, num=15)`

- number of units in a layer is in the range of 0.5x, 1x, 2x and 3x times the number of features (73)
- number of layers: 1, 3, 5 and 9

The best hyper-parameters are an α of 0.3715 and a neural-net configuration of (5, 219) for EvoSuite. For the Randoop testing tool it is 0.002629 and (9, 73).

Testing tool	Time	Math	Lang
"evosuite"	0.255	0.330	0.289
"randoop"	0.168	0.262	0.246

Table 4.5: MAE of the best algorithm Support Vector Regression (SVR) on the Defects4J data. This can be considered test performance.

We can see from table 4.4 that non-linear algorithms Support Vector Regression and Multilayer Perceptron do better than Huber regression as expected. Support Vector Regression consistently performed the best. It achieved MAE values of 0.216 and 0.0880 on validation folds using EvoSuite and Randoop data. It is not clear why algorithms have a smaller error on Randoop data. For the rest of the project Support Vector Regression (SVR) is used for the coverage prediction model. The model is tested on Defects4J and MAE values are presented in table 4.5. The errors are bigger than from cross-validation. A major reason for this is that Defects4J consists of different software projects. The large errors are not as critical for the purpose of ranking. A more detailed analysis is presented in the next chapter.

High rank and soft manners may not always belong to a true heart.

– Anthony Trollope

Chapter 5

Results

5.1 Introduction

In this chapter main results of the simulation are presented. Results are presented in graphical form as well as in the form of statistics. The graphs are primarily AUC-f plots introduced earlier. They are graphed from a single trial, while the statistics are averaged values. First, "preview" statistics are given in order for the reader to get a general idea about the performance. Then more detailed metrics are presented for ranking methods and for the combination method based on coverage prediction and bug density. The previously introduced ranking methods are considered: LOC, BUG, BUG_DENSITY, COVERAGE_PREDICTION, and COVERAGE_DENSITY_COMBO (CDC). There is also a new method called COVERAGE_BRANCH. It is the predicted coverage score from the previously presented COVERAGE_PREDICTION model, but weighted by the total number of Java bytecode branches of a Class under Test (CUT), and sorted in descending order. The COVERAGE_DENSITY_COMBO method is the most sophisticated one among all, since it contains information from essentially every other method considered. This ranking method is investigated with more detailed statistics in the section 5.4 which contains performance at different budgets. There is also a discussion on human inspection effort of buggy classes that are detected and those that are undetected. Next, the CDC method is evaluated with a non-parametric Wilcoxon test at different budgets. In the last section measurements regarding coverage prediction are presented.

5.2 Preview Statistics

In tables 5.1 and 5.2 we can see statistics of different ranking methods put together side-by-side in order to get a sense of the overall performance. For example, $\max \Delta P@20\%$ means it is the maximum deviation in precision achieved by one of the ranking methods by considering the top 20% subset of ranked data. In this context precision refers to the fraction of correct predictions, i.e. the fraction of buggy classes that are detected by the generated tests. This is inspired by the well-known $Precision@K$ metric used for evaluation of IR systems [Zob98]. The difference here is that a relative amount of data is used instead of a fixed K to account for the varying project sizes. The standard test generation budget is 3 minutes (180 seconds) unless otherwise stated. This is modeled in accordance with the experimental setup of Shamshiri et al. [SJR+15].

The first thing that we can note is that bug detection is much higher for EvoSuite than for Randoop. It is in the range of 30%, while it doesn't go above 20% for Randoop. Another important observation is that performance on the Time project is abnormal. For Closure there is only an overall bug detection of a few percent and that project is omitted from further investigation. Such

differences with respect to the work by Shamshiri et al. [SJR+15] are discussed in the next chapter. We can definitely see that some ranking methods do well in the sense that the AUC-f metric is greater than 0, i.e. an improvement over the baseline. However, the tables presented here show only the best and worst results for the purposes of introduction. In the next section there is a more detailed breakdown of performance by project and ranking type.

Project	Bug detection	$\max \Delta P@20\%$	$\min \Delta P@20\%$	$\max \Delta P@40\%$	$\min \Delta P@40\%$
"Time"	0.370	0.0963	−0.104	0.145	0.0236
"Math"	0.352	0.124	−0.0189	0.0684	−0.0189
"Lang"	0.354	0.108	−0.123	0.133	−0.110

Table 5.1: Overall performance for the testing tool EvoSuite. Here, performance is presented by considering the largest and smallest deviations in precision by ranking methods.

Project	Bug detection	$\max \Delta P@20\%$	$\min \Delta P@20\%$	$\max \Delta P@40\%$	$\min \Delta P@40\%$
"Time"	0.111	0.156	−0.0444	0.0404	−0.0202
"Math"	0.204	0.145	−0.0298	0.0496	0.001 95
"Lang"	0.108	0.149	0.0974	0.0974	0.007 69

Table 5.2: Overall performance for the testing tool Randoop. Here, performance is presented by considering the largest and smallest deviations in precision by ranking methods.

5.3 Performance of Ranking Methods

In this section performance of ranking methods is presented in graphical as well as tabular form. The primary metric is again AUC-f. This metric was introduced in section 3.4.2. In the best case scenario we want the cumulative detection curve to be as close as possible to the one by ideal ranking which puts all detectable classes to the front. The graphs give a good sense of the dynamics of ranking. The accompanying statistics give the quantitative aspect behind the curves.

Overall, bug detection is modest and proves to be a challenging task. If we follow the numbers for each ranking method individually, then it does not always perform in a consistent way for every project. Methods based on LOC directly or indirectly through density have comparable performance. COVERAGE_DENSITY_COMBO does consistently well, i.e. never drops below to the last places compared to other methods.

5.3.1 EvoSuite

Time

Ranking method	Bug detection	$\Delta P@20\%$	$\Delta P@40\%$	AUC metric
"BUG"	0.370	0.0296	0.114	0.345
"COVERAGE_BRANCH"	0.370	0.0296	0.0842	0.299
"COVERAGE_PREDICTION"	0.370	0.0296	0.0539	0.187
"COVERAGE_DENSITY_COMBO"	0.370	0.0296	0.0539	0.187
"BUG_DENSITY"	0.370	−0.0370	0.114	0.111
"LOC"	0.370	−0.0370	0.0236	−0.0449

Table 5.3: Detailed performance of ranking methods for the project Time and EvoSuite (sorted by AUC-f). This is one of the occasions when error proneness method dominates.

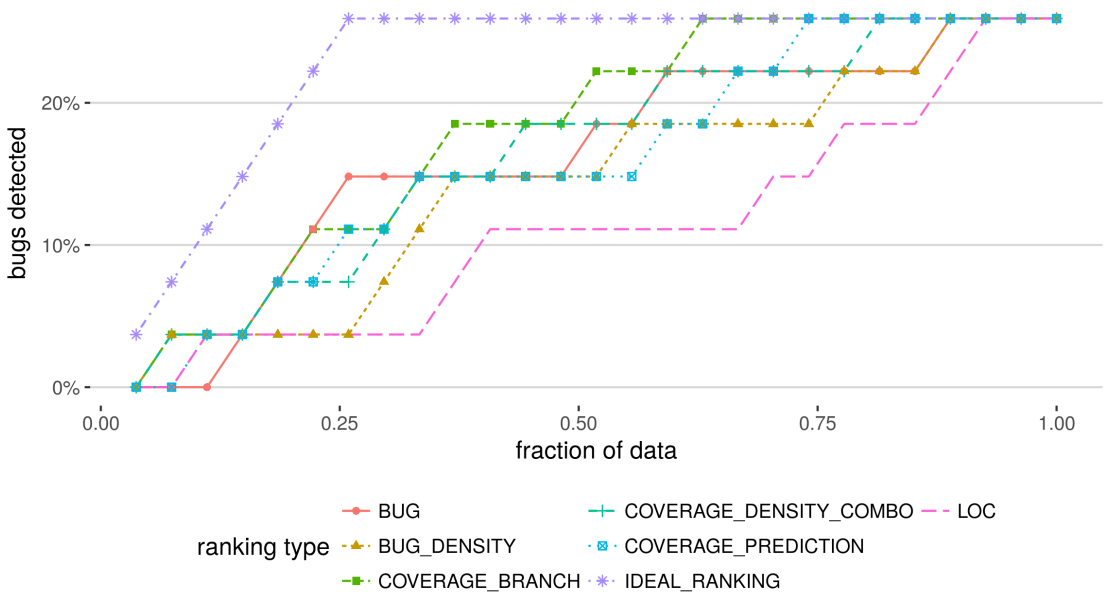

Figure 5.1: *Bug detection as a function of ranked data.* Performance of ranking methods for the Time project and EvoSuite. The BUG ranking method gains an advantage by ordering classes correctly at the front. The AUC-f metric properly captures this.

Math

Ranking method	Bug detection	$\Delta P@20\%$	$\Delta P@40\%$	AUC metric
"COVERAGE_BRANCH"	0.352	0.140	0.0446	0.180
"BUG"	0.352	0.0922	0.0605	0.122
"BUG_DENSITY"	0.352	0.0288	0.0288	0.0900
"COVERAGE_DENSITY_COMBO"	0.352	0.0129	0.0367	0.0823
"LOC"	0.352	0.0605	−0.002 99	0.0458
"COVERAGE_PREDICTION"	0.352	0.0446	−0.002 99	0.0123

Table 5.4: Detailed performance of ranking methods for the project Math and EvoSuite (sorted by AUC-f). It is unexpected that BUG does better than their more advanced counterparts. The AUC-f metric captures overall performance. However, precision at different levels is not always consistent for the same method. COVERAGE_DENSITY_COMBO achieves second to best precision at 40% which is also desirable.

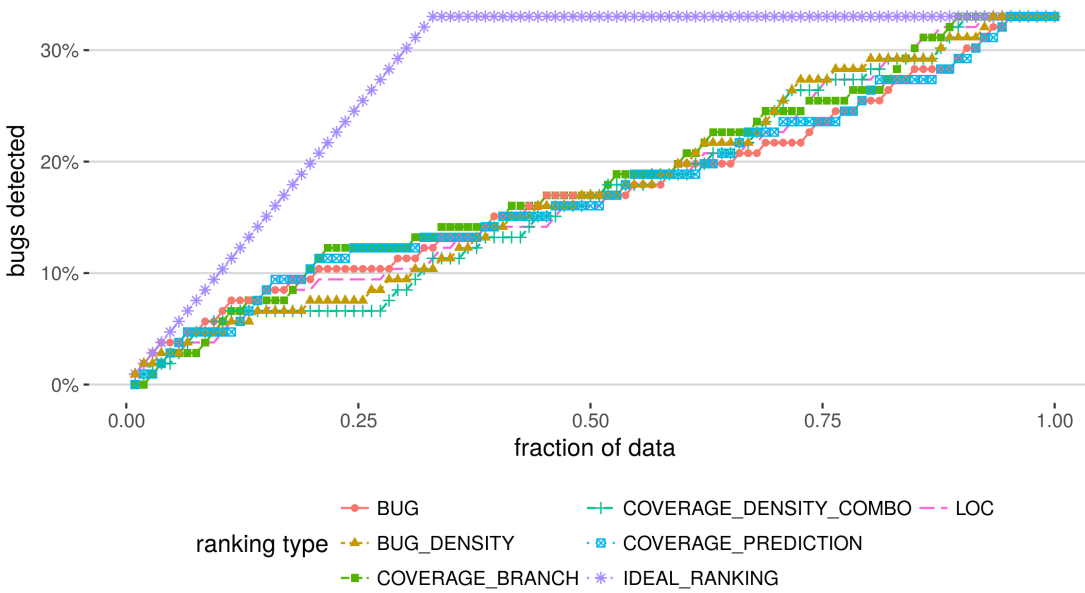

Figure 5.2: *Bug detection as a function of ranked data.* Performance of ranking methods for the Math project and EvoSuite. The COVERAGE_DENSITY_COMBO method performs poorly in the beginning, but makes up towards the end. This could potentially be tweaked with different combination weights, because the COVERAGE_PREDICTION method does better in the beginning.

Lang

Ranking method	Bug detection	$\Delta P@20\%$	$\Delta P@40\%$	AUC metric
"COVERAGE_BRANCH"	0.354	0.0308	0.108	0.282
"LOC"	0.354	0.0821	0.0821	0.262
"COVERAGE_DENSITY_COMBO"	0.354	−0.0462	0.0949	0.250
"BUG_DENSITY"	0.354	0.005 13	0.133	0.212
"COVERAGE_PREDICTION"	0.354	0.0308	−0.0205	0.0866
"BUG"	0.354	−0.0718	−0.0974	−0.147

Table 5.5: Detailed performance of ranking methods for the project Lang and EvoSuite (sorted by AUC-f). Even the simplest model LOC can do well and outperform others. The ranking method BUG_DENSITY has a high AUC-f overall, yet it doesn't have high precision at 20%.

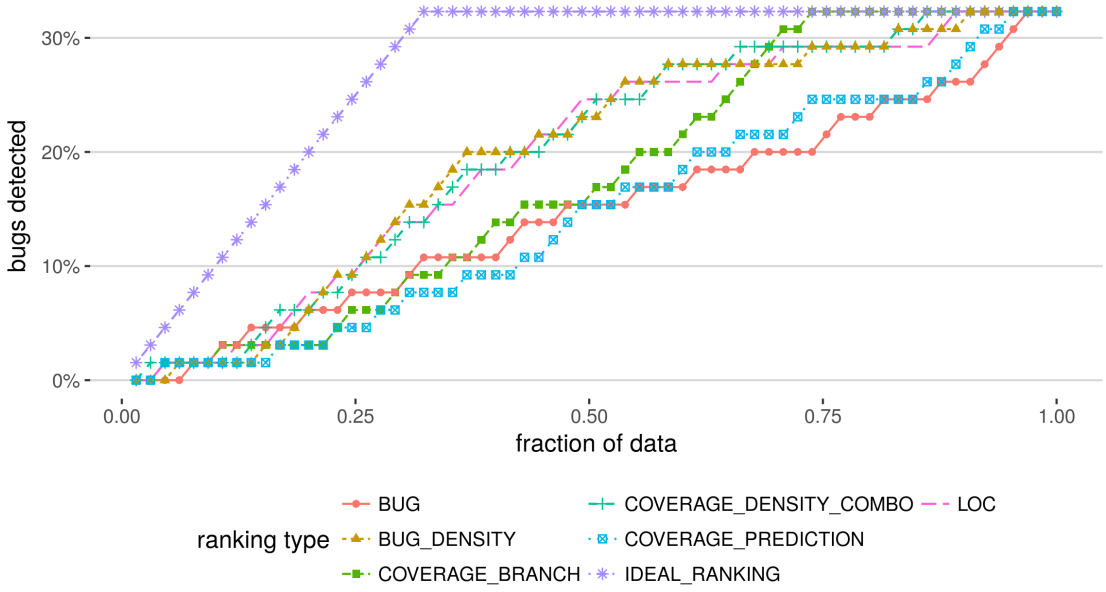

Figure 5.3: *Bug detection as a function of ranked data.* Performance of ranking methods for the Lang project and EvoSuite. This plot illustrates nicely how LOC-related methods BUG_DENSITY and COVERAGE_DENSITY_COMBO follow along a similar path.

5.3.2 Summary of ranking methods performance for EvoSuite

Table 5.6 shows that the best ranking methods are BUG and COVERAGE_BRANCH (twice) for the projects Time, Math and Lang. The AUC-f values achieved are 0.345, 0.180 and 0.282. It is also interesting to see that such a simple method as LOC by itself achieves second place AUC-f for the Lang project. The CDC method is a consistent performer and ranks in the middle among

Project	Ranking method	$\Delta P@20\%$	$\Delta P@40\%$	AUC metric
"Time"	"BUG"	0.0296	0.114	0.345
"Math"	"COVERAGE_BRANCH"	0.140	0.0446	0.180
"Lang"	"COVERAGE_BRANCH"	0.0308	0.108	0.282

Table 5.6: Best ranking methods for EvoSuite according to tables 5.3, 5.4, and 5.5.

other methods. For the project Lang it came in third. The AUC-f metric appears to capture overall dynamics of a ranking method well if the ranking curve is stable, i.e. does not have a lot of curvature. This does not always hold true. In the case of Math the COVERAGE_PREDICTION has a modest AUC-f value of 0.0123, but it has some fluctuation in the beginning and towards the end (see Fig. 5.2). We can also see that the methods BUG and LOC seem to be anti-proportional: when one of them performs highly, the other one performs poorly.

5.3.3 Randoop

Time

Ranking method	Bug detection	$\Delta P@20\%$	$\Delta P@40\%$	AUC metric
"COVERAGE_BRANCH"	0.111	0.222	0.0404	0.600
"BUG"	0.111	0.0889	0.0404	0.398
"COVERAGE_PREDICTION"	0.111	0.0889	-0.0202	-0.0475
"COVERAGE_DENSITY_COMBO"	0.111	-0.0444	-0.0202	-0.189
"BUG_DENSITY"	0.111	-0.0444	-0.0202	-0.275
"LOC"	0.111	-0.0444	-0.0202	-0.336

Table 5.7: Detailed performance of ranking methods for the project Time and Randoop (sorted by AUC-f). There is a very large spread in AUC-f values compared to other settings.

The time project has some of the biggest fluctuations in AUC-f values not only in relative terms, but in absolute terms as well. The largest AUC-f is 0.600, while the smallest is -0.336. Such variability is likely due to the fact that the Time project contains a limited number of datapoints (27) and the bug detection is low. We can also observe that LOC-related methods COVERAGE_DENSITY_COMBO, BUG_DENSITY and LOC perform poorly on this project both in terms of precision and AUC-f. This can also be observed in the appendix fig. A.1.

Math

Ranking method	Bug detection	$\Delta P@20\%$	$\Delta P@40\%$	AUC metric
"COVERAGE_PREDICTION"	0.204	0.0496	0.0496	0.260
"LOC"	0.204	0.145	0.0258	0.195
"COVERAGE_DENSITY_COMBO"	0.204	−0.0139	0.0337	0.187
"BUG_DENSITY"	0.204	0.0178	0.009 88	0.140
"BUG"	0.204	0.0496	0.0258	0.122
"COVERAGE_BRANCH"	0.204	0.0972	0.001 95	−0.001 91

Table 5.8: Detailed performance of ranking methods for the project Math and Randoop (sorted by AUC-f). COVERAGE_DENSITY_COMBO remains a consistent performer.

For this repository LOC as well as LOC-related methods BUG_DENSITY and COVERAGE_DENSITY_COMBO do well. However, COVERAGE_PREDICTION achieves the highest AUC-f value of 0.260. So far the COVERAGE_DENSITY_COMBO method performs consistently well, even though it does have negative precision at 20%.

Lang

Ranking method	Bug detection	$\Delta P@20\%$	$\Delta P@40\%$	AUC metric
"COVERAGE_PREDICTION"	0.108	0.149	0.0974	0.513
"COVERAGE_DENSITY_COMBO"	0.108	0.0974	0.0718	0.463
"BUG_DENSITY"	0.108	0.0974	0.0718	0.356
"LOC"	0.108	0.0974	0.0333	0.312
"BUG"	0.108	0.123	0.007 69	0.199
"COVERAGE_BRANCH"	0.108	−0.0564	−0.005 13	0.0643

Table 5.9: Detailed performance of ranking methods for the project Lang and Randoop (sorted by AUC-f). LOC-related methods BUG_DENSITY and COVERAGE_DENSITY_COMBO do very well.

Here, we can observe again how LOC and LOC-related methods BUG_DENSITY and COVERAGE_DENSITY_COMBO have great performance. On this project the performance of the three methods is very similar in terms of precision and even AUC-f. Still, COVERAGE_PREDICTION is first, while COVERAGE_DENSITY_COMBO has also a comparably high AUC-f value. This is one of the exceptional settings where the methods LOC and BUG don't have opposite performance in terms of AUC-f or overall placement.

5.3.4 Summary of ranking methods performance for Randoop

Among the best ranking methods for Randoop COVERAGE_PREDICTION appeared twice. For the Lang project it achieved an impressive AUC-f of 0.513. This is an indication that the coverage prediction methods are not only competitive, but detect signal in ways different from the other methods. We can also witness similar performance for LOC-related methods for Math and Lang

Project	Ranking method	$\Delta P@20\%$	$\Delta P@40\%$	AUC metric
"Time"	"COVERAGE_BRANCH"	0.222	0.0404	0.600
"Math"	"COVERAGE_PREDICTION"	0.0496	0.0496	0.260
"Lang"	"COVERAGE_PREDICTION"	0.149	0.0974	0.513

Table 5.10: Best ranking methods for Randoop according to tables 5.7, 5.8, and 5.9.

in tables 5.8 and 5.9. For project Lang COVERAGE_DENSITY_COMBO comes in second place, and ranks competitively for other projects. This is in contrast to COVERAGE_BRANCH method which came in first for the Time project, but last for the other two projects.

5.4 Performance at Different Budgets

5.4.1 Cost plot description

To see whether smaller test generation budgets can be used for the purpose of bug detection additional experiments are carried out. This time different test generation budgets are used. The performance is visually represented in a "cost plot". Ranking curves for the COVERAGE_DENSITY_COMBO method are now considered at different budgets and drawn together on a single plot. The difference is that the x-axis is not project data, but total test generation time. For example, for project Lang consisting of 65 bugs the total test generation time is $65 \cdot 3min = 195min$. The budgets considered are multiples of each other: 45, 90 and 180 seconds. This means that if we put those ranking curves on the same plot, then the three curves would cover a different fraction of the x-axis: 25%, 50% and 100%. In this type of plot the x axis is on a log-scale to make it easier to read. The unit for the x-axis is chosen in such a way that log_2 of the total generation time at a budget of 45 seconds is equal to 1. From this follows that the total generation time at other budgets is equal to 2 and 4. This brings all projects to the same units regardless of number of classes.

The plot allows us to see bug detection at different budgets. By plotting the CDC ranking method curve with the ideal curve it is easier to see the difference. The purpose behind "normalization" of the test generation time is to give a sense of the difference in code testability of different projects. Here, we assume that ranking methods put detectable faulty classes to the front. This is supported by high precision at the top 20% and 40% of data as shown in the previous sections. For the case of standard budget we want additional bug detection to happen in the beginning resulting in higher $\Delta P@20\%$. We want the curves at higher budgets to cross the curves at lower budgets early on. If this doesn't happen, then this can be an indication that the project has "complex" code, because AST performance doesn't improve in the subset of interest.

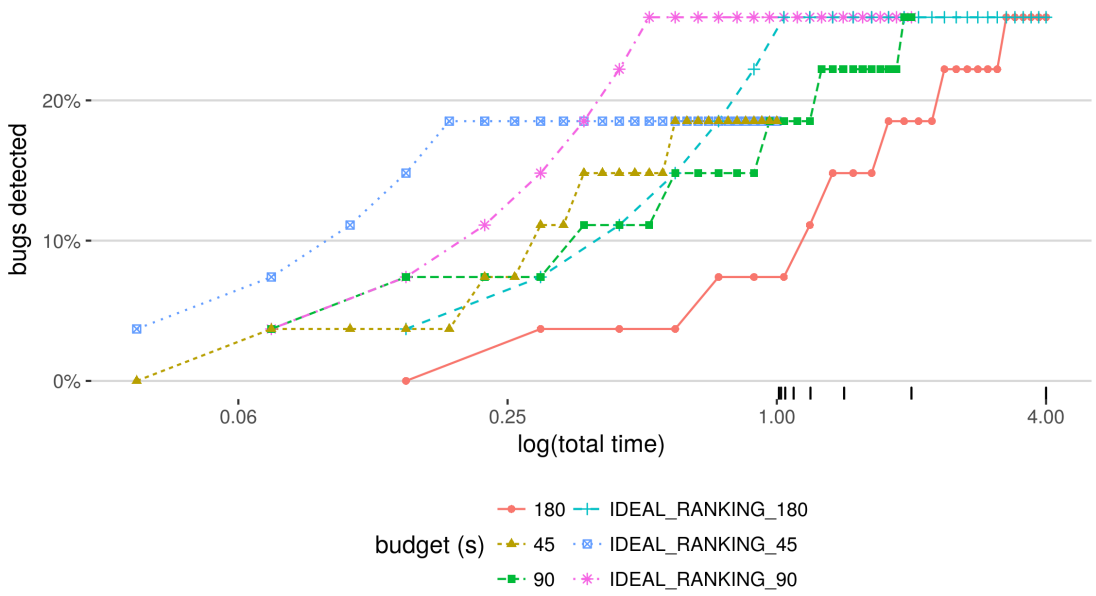

Figure 5.4: *Total generation cost for bug detection.* Performance of COVERAGE_DENSITY_COMBO ranking at different budgets for the Time project and EvoSuite. For the budgets 90s and 180s the bug detection is the same.

5.4.2 EvoSuite

First, we see that bug detection does not differ much at different test generation budgets. The difference is in the 5% range, while bug detection using EvoSuite at the standard budget of 3 minutes is 0.370, 0.352 and 0.354 for the projects Time, Math and Lang. In the first figure 5.4 featuring the Time project the bug detection is even the same for budgets 90s and 180s. It can be even higher for the smaller budget setting as evident from the Math plot in fig.5.5, where the setting at 45s outperforms the setting at 90s. Such occurrences are likely due to the flakiness of tests. Unexpectedly, lower budget curves do better at ranking in the front and achieve higher AUC-f (see Table 5.11) for 2 out of 3 projects.

Project	AUC@45s	AUC@90s	AUC@180s
"Time"	0.395	0.273	0.187
"Math"	0.147	0.0941	0.0826
"Lang"	0.284	0.209	0.252

Table 5.11: AUC metrics for the COVERAGE_DENSITY_COMBO method at different budgets per CUT using EvoSuite.

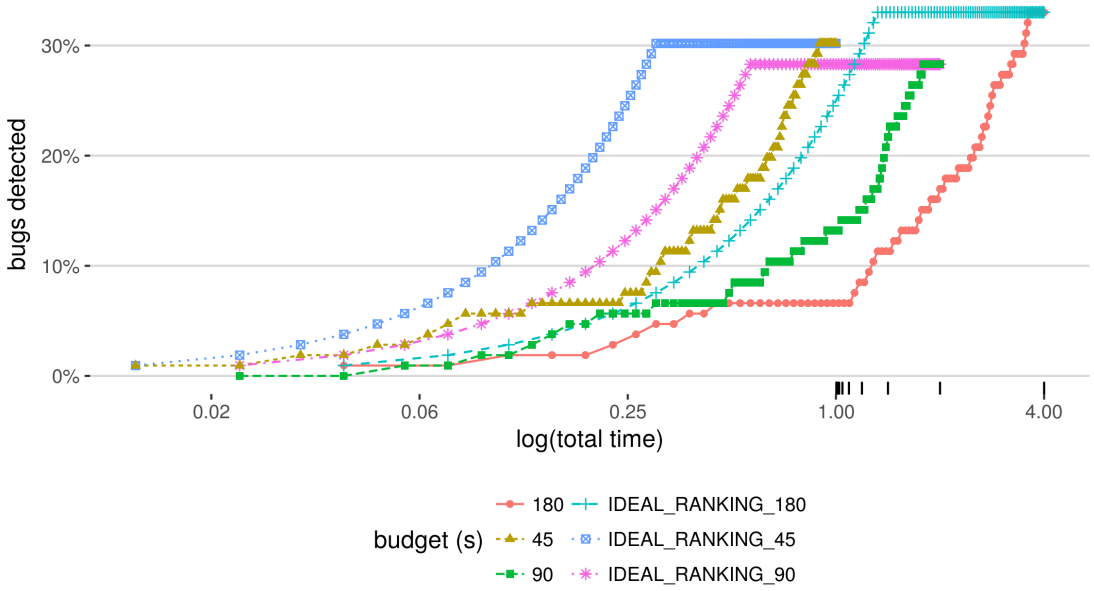

Figure 5.5: *Total generation cost for bug detection.* Performance of COVERAGE_DENSITY_COMBO ranking at different budgets for the Math project and EvoSuite. An even bigger abnormality is when for the generation budget of 45s the bug detection is higher than for the generation budget of 90s.

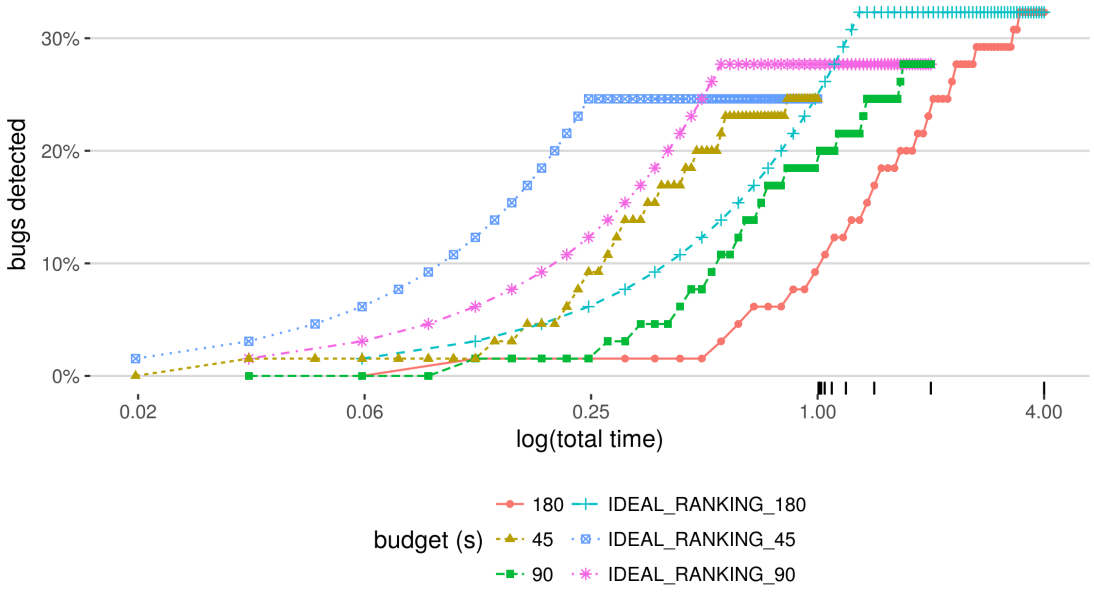

Figure 5.6: *Total generation cost for bug detection.* Performance of COVERAGE_DENSITY_COMBO ranking at different budgets for the Lang project and EvoSuite.

Project	AUC@45s	AUC@90s	AUC@180s
"Time"	NaN	−0.314	−0.189
"Math"	0.296	0.239	0.187
"Lang"	0.466	0.343	0.460

Table 5.12: AUC metrics for the COVERAGE_DENSITY_COMBO method at different budgets per CUT using Randoop. For the Time project at 45s there is only one bug detected which is causing numerical integration problems, so the result is omitted.

5.4.3 Randoop

For the test generation tool Randoop the plots (see appendix section A.2) are less useful due to low bug detection. At the standard budget Randoop has bug detection values of 0.111, 0.204, and 0.108 for the projects Time, Math and Lang. For the low-detection projects Time and Lang the plots look jagged resembling step functions. Thus, we should not draw big conclusions based simply on the numbers for those projects in table 5.12. For the Math project on the other hand we can see in the plot 5.7 behavior resembling that under EvoSuite, where at smaller budgets greater $P@20\%$ is achieved.

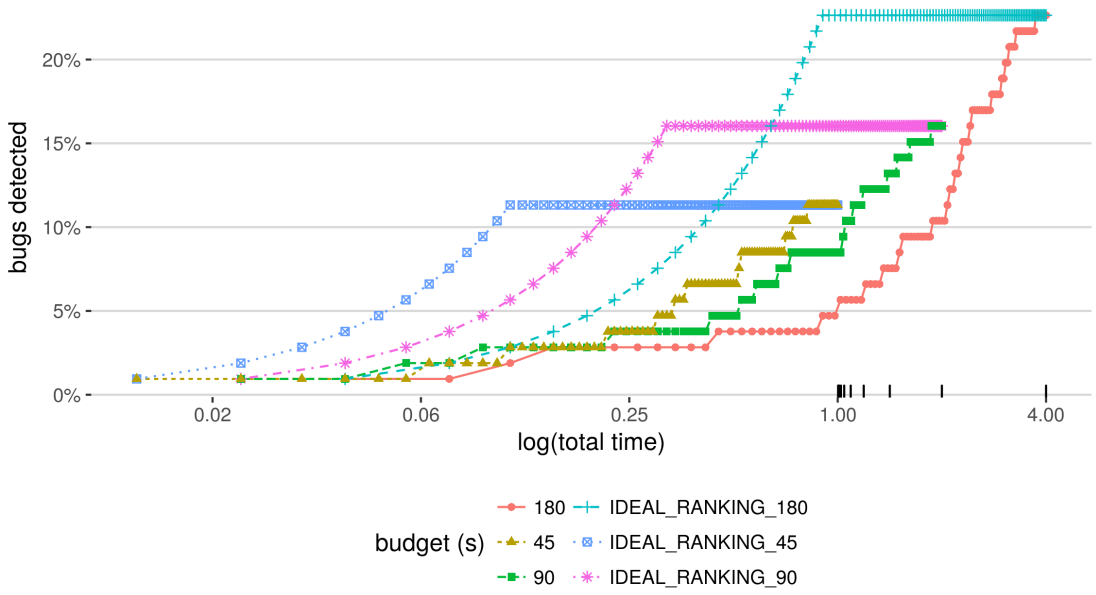

Figure 5.7: *Total generation cost for bug detection.* Performance of COVERAGE_DENSITY_COMBO ranking at different budgets for the Math project and Randoop. Randoop has more pronounced bug detection difference at different test generation budgets. It goes as high as 5%.

5.4.4 Summary

We can see from the plots that undeclared exceptions are detected just as well at smaller budgets. There is plenty of evidence that smaller test generation budgets can be recommended for the same

task. Furthermore, the ranking appears to be just as stable at different budgets as indicated by the positive AUC-f metric. Sometimes bug detection at a higher budget is smaller than the one at a lower budget. This is likely due to the filtering process of flaky tests which come about due to the randomized nature of algorithms employed in test generation tools.

5.5 Human Inspection Effort of Bugs

Project	EvoSuite LOC det.	EvoSuite LOC undet.	Randoop LOC det.	Randoop LOC undet.
"Lang"	474.6	713.0	452.4	647.7
"Math"	322.3	399.1	280.6	394.6
"Time"	512.3	571.2	732.4	526.0

Table 5.13: *LOC averages for detected and undetected bugs.* In all scenarios detected bugs have a smaller number of LOC. Only in the case of the project Time and Randoop is the average LOC higher for detected bugs.

In previous studies on bug prediction LOC of classes has been used as a proxy for human inspection effort [CLP+15, PAP+16]. Here, the average LOC of detected and undetected buggy classes are presented. Table 5.13 shows macro-averages over 3 trials with data for EvoSuite and Randoop side-by-side. In five out of six cases the LOC is lower for detected buggy classes. Only for the project Time and tool Randoop is the average higher. Overall, there is strong evidence that the mean is lower for detected bugs. It is most pronounced in the Lang project where it is about 1.5 times smaller.

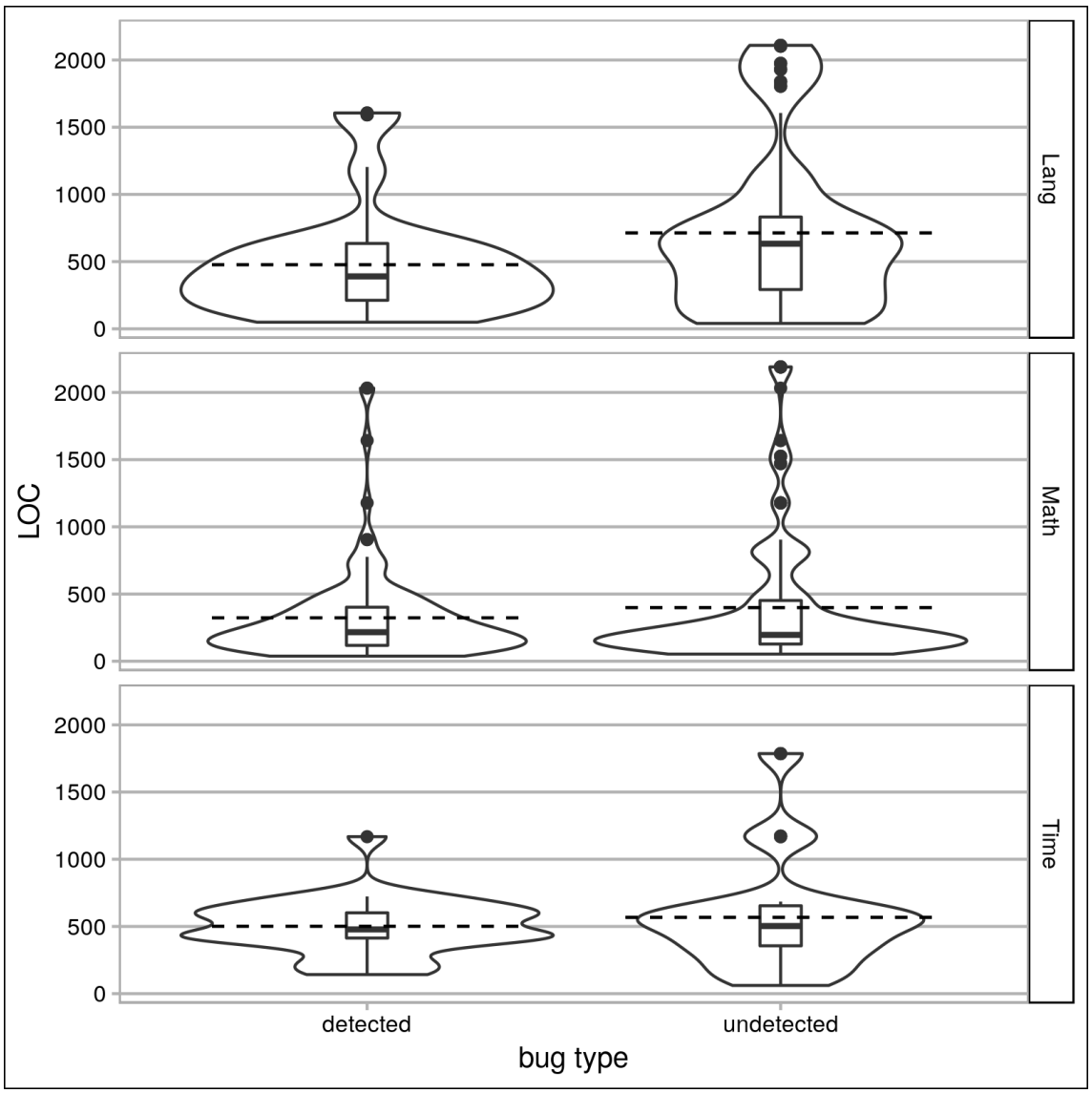

Figure 5.8: *Violin and box plots for LOC of detected and undetected bugs in EvoSuite.* The dashed line represents the mean for easier comparison. Even though the means are lower for detected bugs the distributions are fairly similar with many outliers also in the detected category.

A more detailed comparison is presented in figures 5.8 and 5.9 with violin and box plots where the data is micro-averaged. A general observation is that the median LOC for classes from Defects4J is under 500. We can see that in the case of EvoSuite the distribution of class sizes is almost identical between the two categories of classes including the presence of outliers. We cannot say the same for Randoop (fig. 5.9). For that tool the distribution is similar only for the Math project. Considering the low number of trials, the skewed distribution and the need for a multiple mean comparison it doesn't make sense to apply a statistical test here.

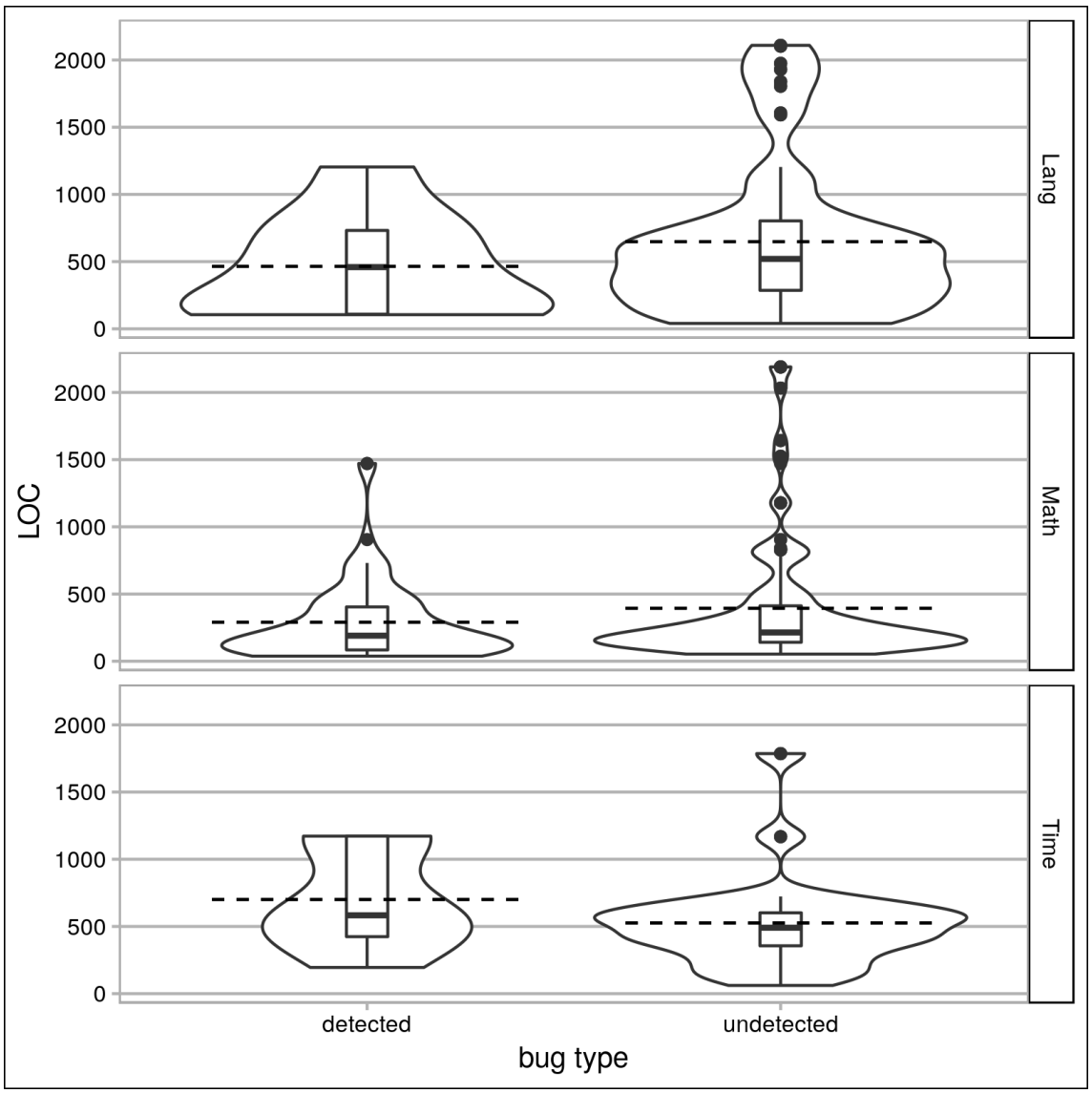

Figure 5.9: *Violin and box plots for LOC of detected and undetected bugs in Randoop.* The dashed line represents the mean for easier comparison. The means are lower in the detected class in Lang and Math, but the distributions look quite different.

5.6 Statistical Tests

To address the first research question Wilcoxon tests with the alternative hypothesis $\mu > 0$ are presented in table 5.14 and 5.15. We'd like to see whether the CDC ranking method is effective by measuring its AUC-f value. Together with the low number of trials (3) the results are not statistically significant. The standard deviations are prohibitively large. The fact that Wilcoxon non-parametric tests require that the distribution is symmetric is not a big problem here relative to the overall experimental setup. The hypothesis testing performed here is rather simplistic. A

more complicated statistical test setup taking into account independent variables like the testing tool would make it even harder to attain statistical significance.

5.6.1 Tests at standard budget (180s)

Project	AUC mean	AUC std. dev.	p-value
"Time"	0.187	0.216	0.250
"Math"	0.0826	0.0212	0.125
"Lang"	0.252	0.115	0.125

Table 5.14: Wilcoxon tests with the alternative hypothesis $\mu > 0$ for the COVERAGE_DENSITY_COMBO method and EvoSuite.

Project	AUC mean	AUC std. dev.	p-value
"Time"	−0.189	0.288	0.875
"Math"	0.187	0.120	0.125
"Lang"	0.460	0.269	0.125

Table 5.15: Wilcoxon tests with the alternative hypothesis $\mu > 0$ for the COVERAGE_DENSITY_COMBO method and Randoop at a budget of 180s.

5.6.2 Tests at smaller budget (90s)

The second research question is regarding a smaller test budget. The tests show a similar picture (tables 5.16 and 5.17). The same tests were performed when the test generation budget is cut in half resulting in 90 seconds. Even though the mean is above 0 the standard deviation is high. However, there is evidence that bug detection doesn't suffer significantly as a result of lower test generation budget as shown in the previous section. From a practical standpoint this is encouraging, because organizations can reap the benefits of AST with less resources than what is currently considered as a good default.

Project	AUC mean	AUC std. dev.	p-value
"Time"	0.273	0.144	0.125
"Math"	0.0941	0.0635	0.125
"Lang"	0.209	0.0905	0.125

Table 5.16: Wilcoxon tests with the alternative hypothesis $\mu > 0$ for the COVERAGE_DENSITY_COMBO method and EvoSuite at a budget of 90s.

Project	AUC mean	AUC std. dev.	p-value
"Time"	−0.314	0.367	0.875
"Math"	0.239	0.0675	0.125
"Lang"	0.343	0.128	0.125

Table 5.17: Wilcoxon tests with the alternative hypothesis $\mu > 0$ for the COVERAGE_DENSITY_COMBO method and Randoop at a budget of 90s.

5.7 Branch Coverage Ranking

In this section the quality of the ranking methods with respect to branch coverage is presented. For this purpose Java bytecode branch coverage is used as a goal for the AUC-f plot. This type of coverage is one of the standard types of coverage metrics and has been used as a complexity measure in the study by Campos et al. [CAFA14] The plot is analogous to the bug detection plot: it is a cumulative plot that shows the total branch coverage in the top right corner. Ideal ranking of classes is such that classes with the most covered branches come first. This way the total coverage is achieved earlier.

To better analyze the potential for coverage prediction a new ranking method is introduced: COVERAGE_BRANCH. It is the predicted coverage score from the previously presented COVERAGE_PREDICTION model, but weighted by the total number of Java bytecode branches of a Class under Test (CUT). This weighting is more fair, because it considers the size of the class. A larger class with a small percentage of covered branches will be ranked higher as long as the absolute number of predicted covered branches is high. As a result this model achieves a higher AUC-f score. Another different ranking method is LOC_DESC, i.e. ordering of classes according to LOC, but in descending order which is the opposite of what was done previously for the LOC ranking method.

Project	Bug detection	Branch coverage	CB	LOC_DESC	CP	BUG	CDC
"Time"	0.407	0.802	0.686	0.834	0.260	0.779	−0.445
"Math"	0.358	0.661	0.657	0.792	0.0464	0.649	−0.479
"Lang"	0.354	0.480	0.800	0.415	0.460	0.0244	−0.365

Table 5.18: Performance of ranking methods for EvoSuite. The AUC-f metrics are for the goal of branch coverage. The initialisms used are for the following ranking methods: COVERAGE_BRANCH (CB), LOC descending (LOC_DESC), COVERAGE_PREDICTION (CP), number of BUGs prediction (BUG), and COVERAGE_DENSITY_COMBO (CDC).

In tables 5.18 and 5.19 AUC-f metrics of ranking methods are presented. We can see that overall EvoSuite achieves higher branch coverage than Randoop. There is a correlation between coverage and overall bug detection. It should be noted that coverage is a necessary, but not sufficient condition for bug detection. We can also observe that even COVERAGE_PREDICTION method performs better with EvoSuite the derived COVERAGE_BRANCH method performs about equally well under both tools. The COVERAGE_BRANCH method does very well when used with both tools. Its performance is comparable to that of LOC_DESC. This can be witnessed in detail in the graphs below.

Project	Bug detection	Branch coverage	CB	LOC_DESC	CP	BUG	CDC
"Time"	0.0741	0.325	0.728	0.744	−0.302	0.641	−0.555
"Math"	0.132	0.352	0.870	0.807	0.142	0.681	−0.496
"Lang"	0.123	0.658	0.841	0.943	0.0872	0.160	−0.869

Table 5.19: Performance of ranking methods for Randoop. The AUC-f metrics are for the goal of branch coverage. The initialisms used are for the following ranking methods: COVERAGE_BRANCH (CB), LOC descending (LOC_DESC), COVERAGE_PREDICTION (CP), number of BUGs prediction (BUG), and COVERAGE_DENSITY_COMBO (CDC).

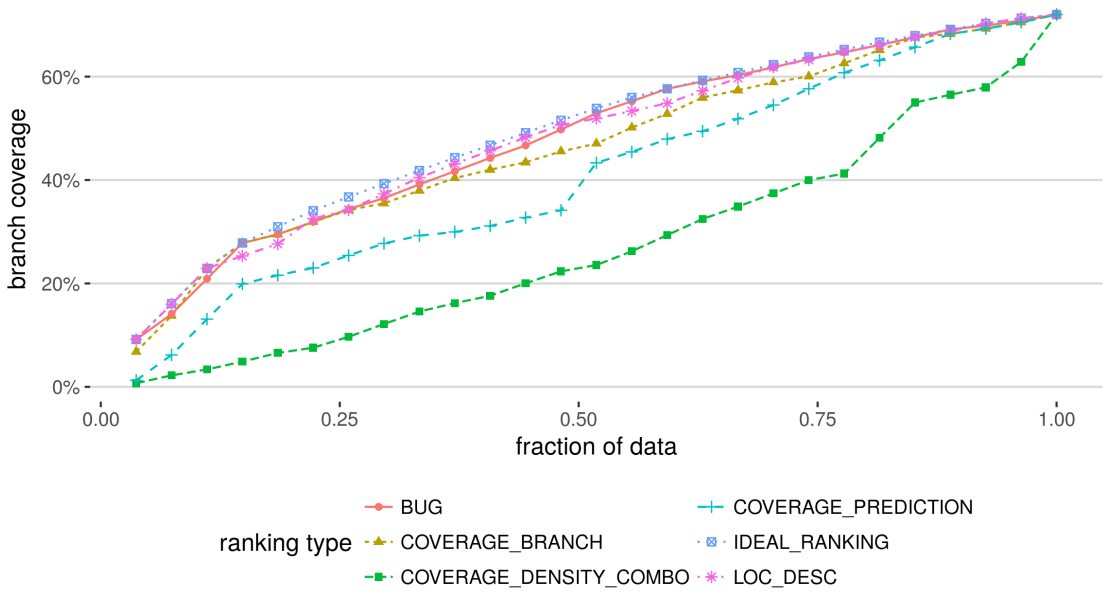

Figure 5.10: *Branch coverage as a function of ranked data.* Branch coverage ranking for the project Time and EvoSuite. Note that BUG and LOC_DESC follow a similar path.

The COVERAGE_DENSITY_COMBO ranking method is the worst when it comes to maximizing branch coverage. This is likely due to the fact that it is influenced by the LOC method. On the other hand, the LOC_DESC method ,which orders classes in descending class size order, does very well. Together with the COVERAGE_BRANCH method they are very close to the ideal ordering. Surprisingly, the bug prediction method does very well and achieves high AUC-f. If you look closer, you will notice that it resembles the LOC_DESC curve. This is a plausible explanation to the mystery: higher error-proneness score is assigned to large classes. Finally, the COVERAGE_PREDICTION method has a modest performance here, and even overtakes the BUG curve in fig. 5.12. The situation for Randoop is similar which can be seen in graphs in the appendix section A.4.

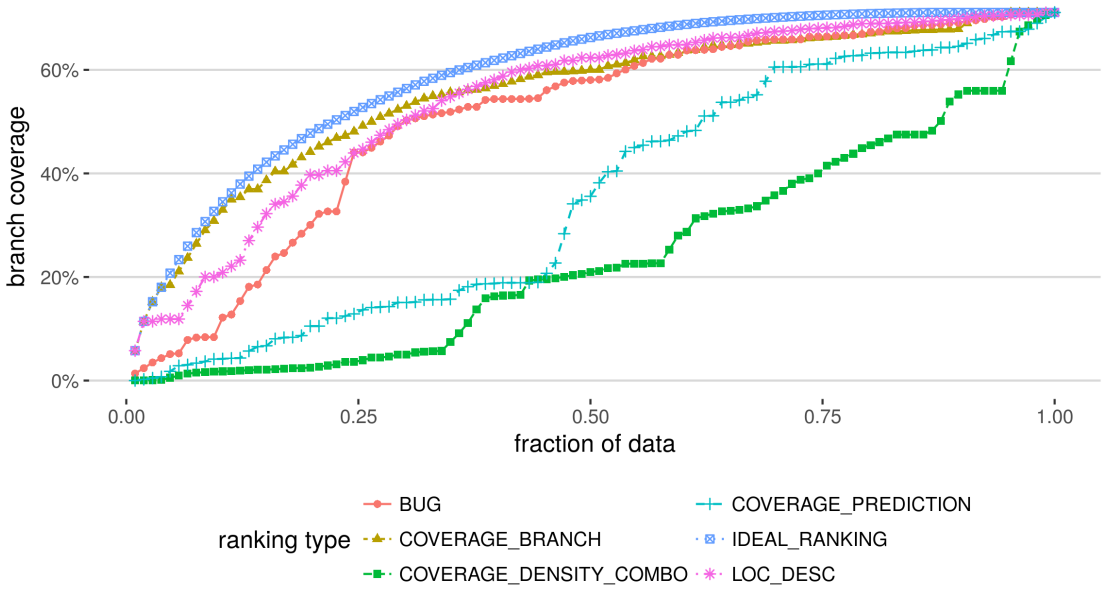

Figure 5.11: *Branch coverage as a function of ranked data.* Branch coverage ranking for the project Math and EvoSuite. The kinks in the COVERAGE_PREDICTION curve are not reflected in the COVERAGE_DENSITY_COMBO curve. The BUG and LOC_DESC methods follow a similar path.

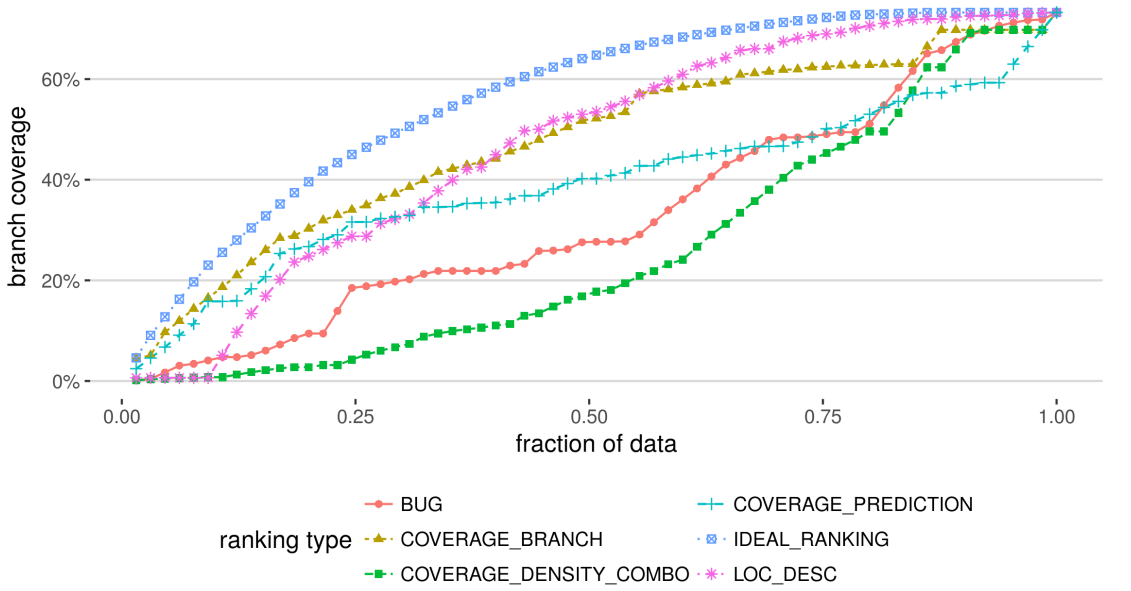

Figure 5.12: *Branch coverage as a function of ranked data.* Branch coverage ranking for the project Lang and EvoSuite. This is one of the few cases where the BUG method does not follow closely the LOC_DESC curve.

Chapter 6

Discussion

6.1 General Remarks

Bug detection is a challenging task. For EvoSuite it is the 30% range, but lower than 20% for Randoop. There is a big difference in bug detection in this thesis and the work by Shamshiri et al. They report bug detection that goes as high as 50% something that was never observed here. The most likely reason for this lies in the filtering of failing and flaky tests: "each compilable test suite was executed on the fixed version five times. If any of these executions revealed flaky tests, then these tests were removed, and the test suite re-compiled and re-executed." [SJR+15] The functionality for fixing test suites in Defects4J is nearly identical [Jus]. It remains unclear why there is a discrepancy in final results.

Most of the analysis is focused on evaluating the quality of ranking. For this purpose the AUC-f metric was introduced. It would have been nice to do the analysis by varying the test generation budget according to the prediction scores by one of the methods. However, it took 4 server days to build the prediction model, and 6 days to run the simulation. Doing a fine-grained simulation with a test generation step-size of 20 seconds increases the running time by an order of magnitude. This alone would require engineering to execute code on a cluster.

6.2 Viability of Ranking Methods

The results look encouraging. The COVERAGE_DENSITY_COMBO method placed consistently in the middle among other methods. In the five out of six times it achieved a positive AUC-f metric. Its relative method COVERAGE_PREDICTION on the other hand was the best performer two times on the Randoop data. This may be an indication that the combination in the form of a harmonic mean still gives too much weight towards BUG_DENSITY which in turn is affected by LOC. This may also explain why the method doesn't perform as well with the goal of branch coverage ranking. Overall, the method appears to be useful in terms of adequate performance, and it is consistent in the application of bug detection. Consistency appears to be its strength. In contrast, the more traditional coverage-oriented COVERAGE_BRANCH method achieved first place in 3 out of 6 cases, but also last place in 2 out of 6 cases.

We have seen that LOC-related methods tend to have similar performance both in terms of numbers as well as following along the same path on the AUC-f plot. According to results on branch coverage it appears that even the BUG method is heavily influenced by LOC, however it is sorting in the opposite direction: the bigger the class the more error-prone it is considered. Intuitively, it makes more sense to have LOC-based methods to order in ascending order only. The rationale is that a smaller class is more thoroughly tested under the same resource allocation.

We should be critical to the fact that the BUG method was the best only on the Time project. In the majority of cases we can observe anti-proportional behavior with the LOC method. The information from LOC alone appears to be dominating the scores whether it is in the form of density or class size like for BUG. This supports the coverage prediction method presented in this thesis as a new and different type of method.

6.3 Code Coverage Importance

The coverage prediction model by itself has a high MAE in the range of 0.09-0.26 for cross-validation, and 0.17-0.33 on the Defects4J dataset. The error is prohibitively large for the model to be used effectively as a developer-assisting IDE plugin. Also, it is interesting that MAE is lower for Randoop data. A possible explanation for this is that Randoop behavior is easier to predict. In this thesis extensive feature engineering was performed with three different algorithms. It's not obvious how this part can be further improved. Increasing training data by orders of magnitude can potentially help improve the model. The SF100 corpus by Fraser et al. [FA15] is a candidate for such training data.

The high MAE did not seem to be a problem for ranking purposes. The COVERAGE_BRANCH method performed great at the task of branch coverage ordering and bug detection. The BUG method was another method that did well at this task. Both of them performed well at the task of ordering classes by bug detection. This was partially made possible by the fact that Defects4J buggy classes are relatively small with the median LOC at 500. This explains why even the LOC_DESC method did well at branch coverage ranking. If the class sizes were more stratified beyond 2000 LOC, then even the best test generation tool wouldn't be able to easily attain good coverage. We can expect that COVERAGE_BRANCH would achieve a pronounced advantage over methods like BUG and LOC_DESC in such case.

The findings regarding human inspection effort of detectable bugs is quite far-reaching. There is evidence that the average LOC of bugs detectable by AST is lower. By itself this is good news, because LOC is a proxy for inspection effort [CLP+15, PAP+16]. We already know that coverage is a necessary, but not sufficient condition for bug detection. The fact that detectable bugs have a lower LOC is additional support for LOC and density based ranking methods that sort classes in increasing order. The bigger lesson here is for the Continuous Test Generation (CTG) paradigm: maximizing code coverage and bug detection are somewhat opposite goals. The first is achieved by allocating the most resources to the largest classes in order to gain more coverage for the project globally. On the other hand, in order to get value out of those automatically generated tests we'd like to detect bugs. This requires allocation of resources to the smallest classes. Finding the exact balance between the two goals is a topic for additional research, and we can expect it to be affected by many variables such as the distribution of class sizes, severity of bugs, etc.

Chapter 7

Threats to Validity

Threats to validity are similar to those in the work by Shamshiri et al. [SJR+15]. In this thesis only three projects from Defects4J are analyzed. These projects are written in Java and open-source. Project properties and the kinds of bugs that occur in them may not generalize. Two test generation tools were used. Even though the generation approaches considered here are different none of the tools are based on Dynamic Symbolic Execution, which is another big class of tools.

The presence of unchecked exceptions is treated as bug detection and was not manually verified. Instead, there is a complete reliance on the validity of manual checks done on the same data by the authors in the related study on Defects4J [SJR+15]. In papers on AST it is common to use CPU time for resource allocation of test generation. This makes it hard to compare results between studies, since virtualization, OS and especially hardware makes things different. Fraser et al. [FA15] gave a reason for using CPU time. It has to do with tool configuration parameters that affect the test generation procedure such as class cast transformation. "In such a context, when different algorithms and variants are compared, is hence important to use as stopping condition the same amount of time (e.g., two minutes per search), and not a fixed number of fitness evaluations." [FA15]

There is a reliance on Defects4J. It is of great quality and has no real alternatives. However, the bugs are not grouped according to severity and all of them were originally selected to be reproduceable by a manually written test. The Defects4J database of bugs enforces a specific experimental setup where the buggy version is manually derived by the authors [JJE14] from the fixed version instead of using the actual commit where the bug was introduced. Another disadvantage of Defects4J is that this is not the only way for regression bugs to be introduced. One such example is when a regression is identified by a developer before committing it.

Another threat is that the error-proneness model used here [OGNL16] is based on immediate history prior to a commit with a buggy class from Defects4J. The Defects4J database shapes the experimental setup in such a way that the time dimension (history) of bugs is ignored. This is not a problem for metrics-based ranking methods such as LOC, COVERAGE_PREDICTION and COVERAGE_BRANCH. However, when constructing a "global" ranking spanning the entire repository history the "local" history scores near the Defects4J bugs are not perfect. A related threat is regarding the same class appearing in multiple bugs. This was mitigated by omitting all Defects4J buggy classes from error-proneness model training.

The experiments were repeated only 3 times, because they are very time-consuming. This simulation required 6 server days to run in addition to 4 server days to build the coverage prediction model. The high standard deviation of measurements is a challenge to attain statistical significance. Just like with most machine learning algorithms large amounts of data would make it even better which is a consideration for future work.

Chapter 8

Conclusion

8.1 Research Questions Revisited

Testing tool	Time	Math	Lang
"evosuite"	0.187	0.0823	0.250
"randoop"	−0.189	0.187	0.463

Table 8.1: AUC-f metrics for the COVERAGE_DENSITY_COMBO method at a test generation budget of 180s.

RQ1. Is the most sophisticated ranking method (combination of coverage prediction and bug density) effective at detecting unchecked exceptions?

The AUC-f metric serves as a measure of effectiveness. The COVERAGE_DENSITY_COMBO (CDC) ranking method was analyzed by giving it a budget of 180 seconds which is modeled after the work of Shamshiri et al. [SJR+15]. The majority of the AUC-f values achieved by the COVERAGE_DENSITY_COMBO (CDC) model are above zero. This is an indication that the ranking method is doing useful work by putting classes under test with unchecked exceptions to the front of the testing queue. The average values of AUC-f metric are presented in table 8.1. Statistical significance for the Wilcoxon tests at a threshold of 0.05 is not achieved, because the standard deviation is very large. This doesn't allow to positively answer the RQ.

Testing tool	Time	Math	Lang
"evosuite"	0.273	0.0944	0.209
"randoop"	−0.314	0.238	0.343

Table 8.2: AUC-f metrics for the COVERAGE_DENSITY_COMBO method at a test generation budget of 90s.

RQ2. Does the most sophisticated ranking method remain effective at a smaller test generation budget?

Here, the answer is also no from a statistical point of view just like in the first question. However,

there is reassuring evidence in the form of aggregated statistics. Wilcoxon tests were applied to AUC-f values obtained under conditions of a test budget one half the original: 90 seconds. The summary from the three trials are in table 8.2. The measurements were also collected at an even smaller budget of 1/4 the standard amount, where the AUC-f values remain to be positive. Another piece of reassuring evidence is that the ranking curves at different budgets appear to have a similar shape without crossings (although do have overlapping points). This can be seen in the figure 8.1 for the Math project to illustrate the point. More such plots are found in the appendix.

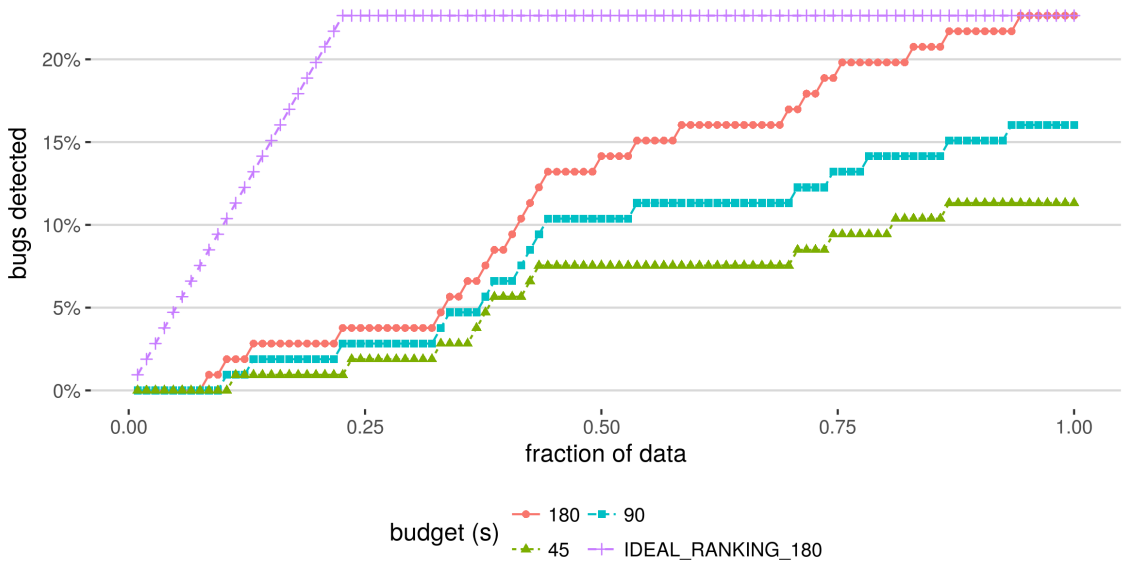

Figure 8.1: Performance of COVERAGE_DENSITY_COMBO ranking at different budgets for the Math project and Randoop. All the curves have similar shapes confirming the expectation that the ranking is stable.

Project	EvoSuite LOC det.	EvoSuite LOC undet.	Randoop LOC det.	Randoop LOC undet.
"Lang"	474.6	713.0	452.4	647.7
"Math"	322.3	399.1	280.6	394.6
"Time"	512.3	571.2	732.4	526.0

Table 8.3: *LOC averages for detected and undetected bugs.* In all scenarios detected bugs have a smaller number of LOC. Only in the case of the project Time and Randoop is the average LOC higher for detected bugs.

RQ3. Is the human inspection effort for detected bugs smaller than for undetected ones?

According to the table 8.3 there is evidence that the bugs in Defects4J have lower LOC and as a result lower human inspection effort. Here, LOC is considered to be a proxy to human inspection

effort just like in previous studies [CLP+15, PAP+16]. Only for project Time and testing tool Randoop it is higher.

8.2 Summary of Contributions

The first main contribution of this thesis is a coverage prediction model for testing tools EvoSuite and Randoop. It can predict the coverage that will be attained before running expensive test generation. This model has been motivated by Campos et al. [CAFA14] for the paradigm of Continuous Test Generation. This is the first known coverage prediction model. It is based on package-level, Chidamber and Kemerer (CK), Object-Oriented (OO) and Java reserved keyword features. To build the model additional training dataset was created based on Apache Cassandra, Apache Ivy, Google Guava and Google Dagger which required collecting compilation dependencies. The model was trained on three machine learning algorithms including Huber Regression, Support Vector Regression and Multi-layer Perceptron. The Mean Absolute Error on the Defects4J datasets by the best algorithm Support Vector Regression is in the range of 0.17-0.33 which is high for the prediction score to be used as a stand-alone metric. However, for the purposes of ranking classes this turned out to be acceptable.

The other major contribution of this thesis is a set of experiments performed on the Defects4J database of bugs. This dataset has been already used previously to show usefulness of Automated Software Testing (AST) by Shamshiri et al. [SJR+15] The simulation performed here is about improving test case prioritization which is reflected in research questions stated above. Part of the purpose is an in-depth evaluation of the coverage prediction model. The following ranking methods were also analyzed: LOC, BUG, BUG_DENSITY, COVERAGE_PREDICTION, COVERAGE_DENSITY_COMBO, and COVERAGE_BRANCH. It was shown that the method based on combination of coverage prediction and error-proneness scores is a consistently performing ranking method. Even though statistical significance was not attained there is evidence that it is effective at standard and lower test generation budgets. Furthermore, the human inspection effort of bugs detected by Automated Software Testing (AST) was measured. This is something that was not performed in the study by Shamshiri et al. [SJR+15]. The average LOC is lower for bugs detectable by Automated Software Testing. This support the ranking methods explored here. This also provides additional insights for the evolution of the Continuous Test Generation (CTG) paradigm, because we would like not only achieve high coverage for a project, but get feedback from tests in the form of unchecked exception discovery.

8.3 Outlook

There is many directions for future work. The results by themselves are useful, since they show viability of ranking methods and the coverage prediction model. This is the first work on coverage prediction, and it appears that the coverage prediction model has some dependency on the testing tool. Ideally, the score would have been general enough to be used as a stand-alone score of complexity in other settings alongside cyclomatic complexity, for example.

Further investigation is required to find out how much coverage prediction is tool-specific and whether one can make generalizations based on three big categories of AST: *random input testing*, *Search-based Software Testing (SBST)* and *Dynamic Symbolic Execution (DSE)*. Another open question is whether approach-specific features can improve MAE of predictions. For example, complex classes with a temporal dependence may not be covered as well by random input testing. Deep learning neural networks may provide improved prediction accuracy. Feature importance scores

by themselves may not be valuable. However, they may give AST tool authors ideas about how to improve their test generation tools. This is especially true for the SBST approach.

The current MAE of the coverage prediction model is high which doesn't matter much for the purposes of ranking. If MAE can be further improved, then the coverage prediction model could be integrated as an IDE plugin. As already discussed rapid releases is becoming the norm. It would be useful for a developer to get feedback on code coverage that can be expected. A company may even have a policy of minimum code coverage for a project. This would prompt the developer to spend writing a test.

An ideal application of the explored ranking methods is useful for a Continuous Integration (CI) server deployment. There is evidence that we are moving towards that considering work on Continuous Test Generation (CTG) [CAFA14]. The sophisticated ranking method from this project is meant to provide faster feedback regarding unchecked exceptions in code. Test generation budget is part of the coverage prediction model. Together with evidence that smaller test generation budgets don't significantly impact bug detection a CTG can be further optimized for computational resource usage. This idea can be seen in fig. 8.1 as well as other plots in the appendix.

Initialisms and Acronyms

AST Automated Software Testing. iii, vii, x, 1, 2, 5, 6, 7, 8, 9, 10, 11, 14, 32, 39, 46, 47, 51

AUC Area under the Curve. ix, 15, 16

AUC-f Area under the Curve fraction. vii, ix, xi, xii, 2, 15, 16, 25, 26, 27, 28, 29, 30, 31, 33, 35, 38, 40, 41, 45, 49

CDC COVERAGE_DENSITY_COMBO. viii, ix, xi, xii, 2, 15, 25, 28, 29, 31, 32, 38, 41, 45, 49, 63, 62, 70

CI Continuous Integration. iii, 1, 2, 9, 11, 52

CK Chidamber and Kemerer. 20

CPU Central Processing Unit. vii, 14, 17, 20, 47

CTG Continuous Test Generation. vii, 2, 10, 46, 51, 52

CUT Class under Test. xi, 10, 15, 25, 33, 40

DIT depth of inheritance tree. 20

DSE Dynamic Symbolic Execution. 5, 9, 47, 51

GA Genetic Algorithm. 5, 11

IDE Integrated Development Environment. 9, 46, 52

IoT Internet of Things. 1

IR Information Retrieval. 21, 25

LOC Lines of Code. vii, ix, xi, xii, 1, 2, 13, 14, 25, 29, 30, 31, 36, 37, 41, 50

MAE Mean Absolute Error. xi, 21, 22, 23, 51

ML Machine Learning. 14

MLR Multi-vocal Literature Review. 7

NOSF number of static fields. 20

NOSI number of static invocations. 20

OO Object Oriented. 20

OS Operating System. 47

QA Quality Assurance. iii, 1, 6, 7

ROC Receiver Operating Characteristic. 16

RQ Research Question. 2, 49, 50

SBST Search-based Software Testing. 5, 51

SUT System under Test. 7

SVR Support Vector Regression. xi, 23

USD United States Dollar. iii

VCS Version Control System. 13

XP Extreme Programming. 5

References

[AFR15] Emil Alégroth, Robert Feldt, and Lisa Ryrholm. Visual gui testing in practice: challenges, problemsand limitations. *Empirical Software Engineering*, 20(3):694–744, 2015.

[AHF+17] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Jānis Benefelds. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, pages 263–272. IEEE Press, 2017.

[AM16] Cyrille Artho and Lei Ma. Classification of randomly generated test cases. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 2, pages 29–32. IEEE, 2016.

[Ani] Maurício Aniche. mauricioaniche/ck: extracts code metrics from java code by means of static analysis. `https://github.com/mauricioaniche/ck`. (Accessed on 07/01/2017).

[Bal98] Thomas Ball. On the limit of control flow analysis for regression test selection. *ACM SIGSOFT Software Engineering Notes*, 23(2):134–142, 1998.

[BCDP+13] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. The evolution of project inter-dependencies in a software ecosystem: The case of apache. In *ICSM*, pages 280–289, 2013.

[BEEGR00] Saida Benlarbi, Khaled El Emam, Nishith Goel, and Shesh Rai. Thresholds for object-oriented measures. In *Software Reliability Engineering, 2000. ISSRE 2000. Proceedings. 11th International Symposium on*, pages 24–38. IEEE, 2000.

[Ber07] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, pages 85–103. IEEE Computer Society, 2007.

[BJC+13] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible debugging software. *University of Cambridge-Judge Business School, Tech. Rep*, 2013.

[BMK04] Jennifer Black, Emanuel Melachrinoudis, and David Kaeli. Bi-criteria models for all-uses test suite reduction. In *Proceedings of the 26th International Conference on Software Engineering*, pages 106–115. IEEE Computer Society, 2004.

[CAFA14] José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. Continuous test generation: enhancing continuous integration with automated test generation. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 55–66. ACM, 2014.

[Cap] Capgemini. wqr_2016-17_final_secure.pdf. https://www.sogeti.com/globalassets/global/downloads/testing/wqr-2016-2017/wqr_2016-17_final_secure.pdf. (Accessed on 07/15/2017).

[CK94] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[CL11] Chih-Chung Chang and Chih-Jen Lin. Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):27, 2011.

[Cla] Mike Clark. clarkware/jdepend: A java package dependency analyzer that generates design quality metrics. https://github.com/clarkware/jdepend. (Accessed on 07/01/2017).

[CLP+15] Gerardo Canfora, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Defect prediction as a multiobjective optimization problem. *Software Testing, Verification and Reliability*, 25(4):426–459, 2015.

[Dan] Al Danial. Aldanial/cloc: cloc counts blank lines, comment lines, and physical lines of source code in many programming languages. https://github.com/AlDanial/cloc. (Accessed on 07/29/2017).

[FA11] Gordon Fraser and Andrea 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.

[FA12] Gordon Fraser and Andrea Arcuri. Sound empirical evidence in software testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 178–188, Piscataway, NJ, USA, 2012. IEEE Press.

[FA15] Gordon Fraser and Andrea Arcuri. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical software engineering*, 20(3):611–639, 2015.

[FN99] Norman E Fenton and Martin Neil. A critique of software defect prediction models. *IEEE Transactions on software engineering*, 25(5):675–689, 1999.

[GBMP13] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[GEM15] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 211–222. ACM, 2015.

[GIP11] Pooja Gupta, Mark Ivey, and John Penix. Testing at the speed and scale of google, 2011.

[GM16] Vahid Garousi and Mika V Mäntylä. When and what to automate in software testing? a multi-vocal literature review. *Information and Software Technology*, 76:92–117, 2016.

[gof] gofraser. Mockito classpath for test generators does not include compilelib · issue #89 · rjust/defects4j. https://github.com/rjust/defects4j/issues/89. (Accessed on 08/17/2017).

[HBB+12] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012.

[HM82] James A Hanley and Barbara J McNeil. The meaning and use of the area under a receiver operating characteristic (roc) curve. *Radiology*, 143(1):29–36, 1982.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.

[HPH+16] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. Comparing white-box and black-box test prioritization. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 523–534. IEEE, 2016.

[HR75] Peter J Huber and Elvezio M Ronchetti. Robustness of design. *Robust Statistics, Second Edition*, pages 239–248, 1975.

[IC] ICS-CERT. Abb multiple components buffer overflow (update) | ics-cert. `https://ics-cert.us-cert.gov/advisories/ICSA-12-095-01A`. (Accessed on 07/15/2017) `https://ics-cert.us-cert.gov/advisories/ICSA-12-095-01A`.

[jdo] jdoop. psycopaths/jdoop: An automatic testing tool for java software. `https://github.com/psycopaths/jdoop`. (Accessed on 09/10/2017).

[JJE14] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 2014.

[Jus] René Just. fix_test_suite.pl. `http://people.cs.umass.edu/~rjust/defects4j/html_doc/fix_test_suite.html`. (Accessed on 08/05/2017).

[JW14] Saint John Walker. Big data: A revolution that will transform how we live, work, and think, 2014.

[JZCT09] Bo Jiang, Zhenyu Zhang, Wing Kwong Chan, and TH Tse. Adaptive random test case prioritization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 233–244. IEEE Computer Society, 2009.

[KNY+15] Mijung Kim, Jaechang Nam, Jaehyuk Yeon, Soonhwang Choi, and Sunghun Kim. Remi: Defect prediction for efficient api testing. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 990–993. ACM, 2015.

[KP02] Jung-Min Kim and Adam Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24th international conference on software engineering*, pages 119–129. ACM, 2002.

[KRTS09] Katja Karhu, Tiina Repo, Ossi Taipale, and Kari Smolander. Empirical observations on software testing automation. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pages 201–209. IEEE, 2009.

[Leh80] Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[LLC+16] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. How does regression test prioritization perform in real-world software evolution? In *Proceedings of the 38th International Conference on Software Engineering*, pages 535–546. ACM, 2016.

[MAK+15] Mika V Mäntylä, Bram Adams, Foutse Khomh, Emelie Engström, and Kai Petersen. On rapid releases and software testing: a case study and a semi-systematic literature review. *Empirical Software Engineering*, 20(5):1384–1425, 2015.

[Mav] Maven. The central repository search engine. `https://search.maven.org/`. (Accessed on 09/06/2017).

[MAZ+15] Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Johannes Gmeiner, and Rudolf Ramler. Grt: An automated test generator using orchestrated program analysis. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 842–847. IEEE, 2015.

[MHS+13] Akito Monden, Takuma Hayashi, Shoji Shinoda, Kumiko Shirai, Junichi Yoshida, Mike Barker, and Kenichi Matsumoto. Assessing the cost effectiveness of fault prediction in acceptance testing. *IEEE Transactions on Software Engineering*, 39(10):1345–1357, 2013.

[MMT+10] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.

[NHC13] Ali Bou Nassif, Danny Ho, and Luiz Fernando Capretz. Towards an early software estimation using log-linear regression and a multilayer perceptron model. *Journal of Systems and Software*, 86(1):144–160, 2013.

[NK15] Jaechang Nam and Sunghun Kim. Clami: Defect prediction on unlabeled datasets (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 452–463. IEEE, 2015.

[OGNL16] Haidar Osman, Mohammad Ghafari, Oscar Nierstrasz, and Mircea Lungu. An extensive analysis of efficient bug prediction configurations. In *Proceedings of the The 13th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE 2017)*. ACM, 2016.

[PAP+16] Annibale Panichella, Carol V Alexandru, Sebastiano Panichella, Alberto Bacchelli, and Harald C Gall. A search-based training algorithm for cost-aware defect prediction. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*, pages 1077–1084. ACM, 2016.

[PE07] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816. ACM, 2007.

[PKT15] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Results for evosuite–mosa at the third unit testing tool competition. In *Search-Based Software Testing (SBST), 2015 IEEE/ACM 8th International Workshop on*, pages 28–31. IEEE, 2015.

[PM17] Annibale Panichella and Urko Rueda Molina. Java unit testing tool competition: fifth round. In *Proceedings of the 10th International Workshop on Search-Based Software Testing*, pages 32–38. IEEE Press, 2017.

[Pra13] IS Wishnu B Prasetya. T3, a combinator-based random testing tool for java: Benchmarking. *FITTEST@ ICTSS*, 8432:101–110, 2013.

[RJGV16] Urko Rueda, René Just, Juan P Galeotti, and Tanja EJ Vos. Unit testing tool competition—round four. In *Search-Based Software Testing (SBST), 2016 IEEE/ACM 9th International Workshop on*, pages 19–28. IEEE, 2016.

[RMPM12] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V Mantyla. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *Proceedings of the 7th International Workshop on Automation of Software Test*, pages 36–42. IEEE Press, 2012.

[RUCH99] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 179–188. IEEE, 1999.

[RUCH01] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering*, 27(10):929–948, 2001.

[SC12] Mark Sanderson and W Bruce Croft. The history of information retrieval research. *Proceedings of the IEEE*, 100(Special Centennial Issue):1444–1451, 2012.

[Sca] Scaleway. Virtual ssd cloud servers - scaleway. `https://www.scaleway.com/virtual-cloud-servers/`. (Accessed on 07/04/2017).

[Sci] ScikitLearn. sklearn.neural_network.mlpregressor — scikit-learn 0.18.2 documentation. `http://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html`. (Accessed on 07/29/2017).

[SGG+14] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. Balancing trade-offs in test-suite reduction. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 246–256. ACM, 2014.

[She88] Martin Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36, 1988.

[SJR+15] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 201–211. IEEE, 2015.

[skl] sklearn. sklearn.svm.svr — scikit-learn 0.19.0 documentation. `http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html`. (Accessed on 09/06/2017).

[SYGM15] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. Comparing and combining test-suite reduction and regression test selection. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 237–247. ACM, 2015.

[WHLA97] W Eric Wong, Joseph R Horgan, Saul London, and Hiralal Agrawal. A study of effective regression testing in practice. In *Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on*, pages 264–274. IEEE, 1997.

[WNT17] Song Wang, Jaechang Nam, and Lin Tan. Qtep: quality-aware test case prioritization. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 523–534. ACM, 2017.

[ZNG+09] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100. ACM, 2009.

[Zob98] Justin Zobel. How reliable are the results of large-scale information retrieval experiments? In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 307–314. ACM, 1998.

Appendix A

Appendix

A.1 Ranking type plots for Randoop

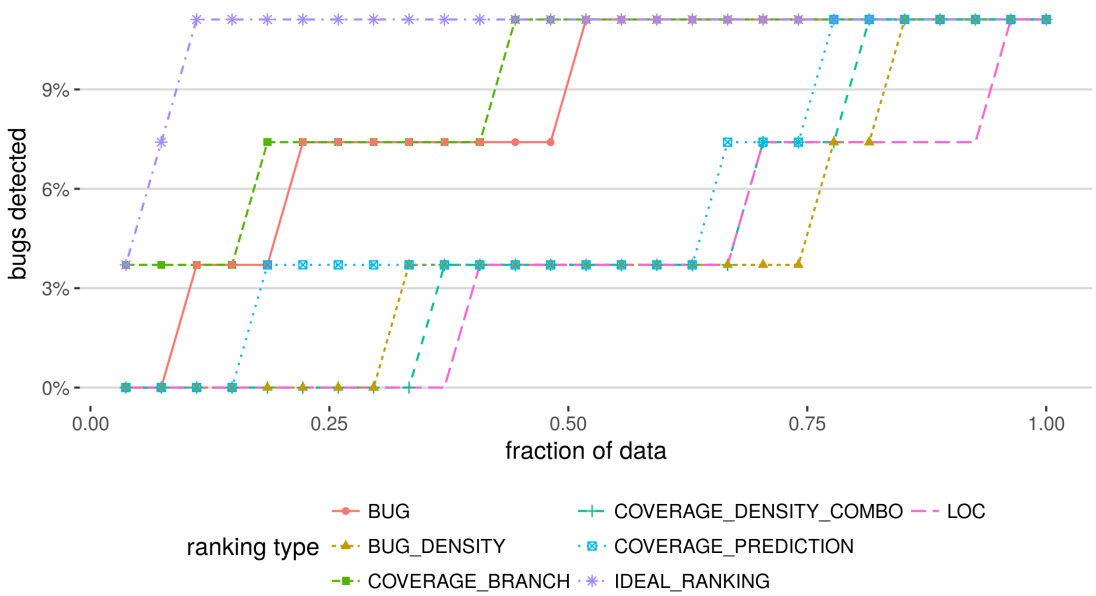

Figure A.1: *Bug detection as a function of ranked data.* Performance of ranking methods for the Time project and Randoop.

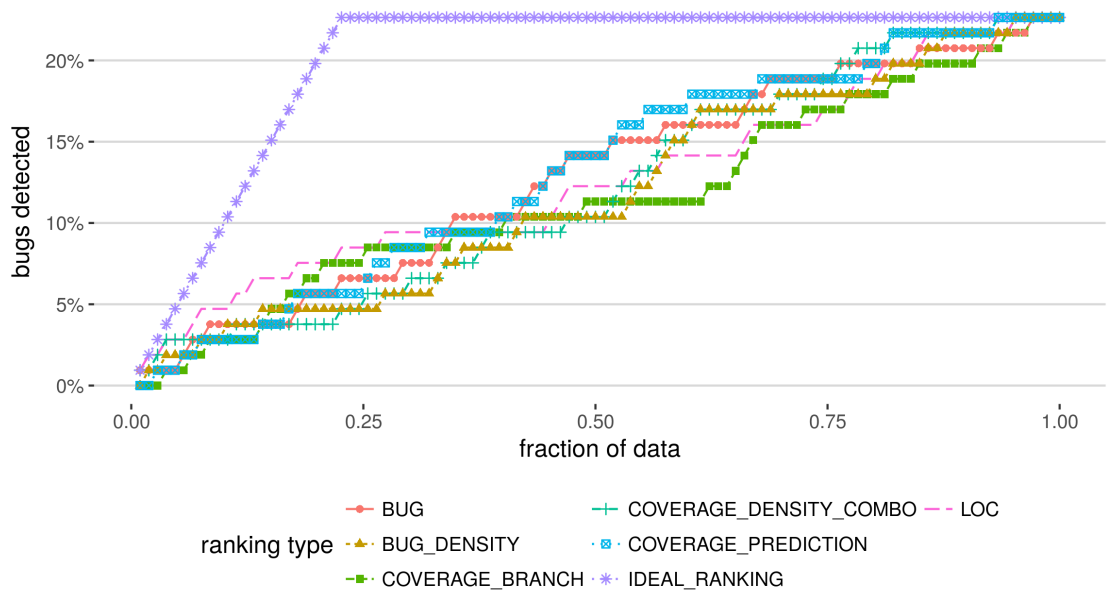

Figure A.2: *Bug detection as a function of ranked data.* Performance of ranking methods for the Math project and Randoop.

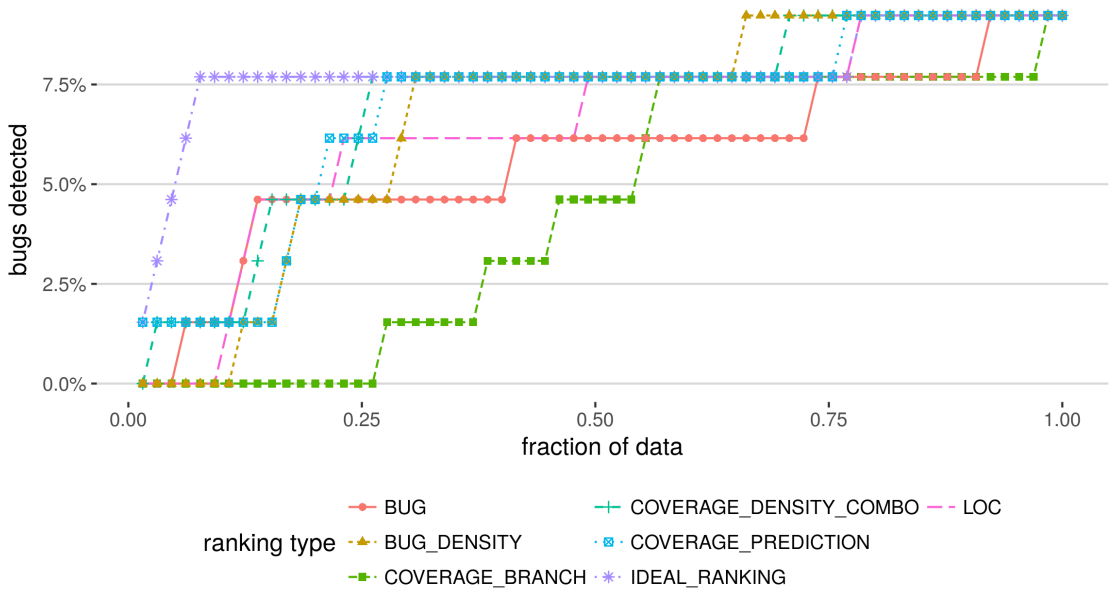

Figure A.3: *Bug detection as a function of ranked data.* Performance of ranking methods for the Lang project and Randoop.

A.2 COVERAGE_DENSITY_COMBO plots for Randoop (log-scale)

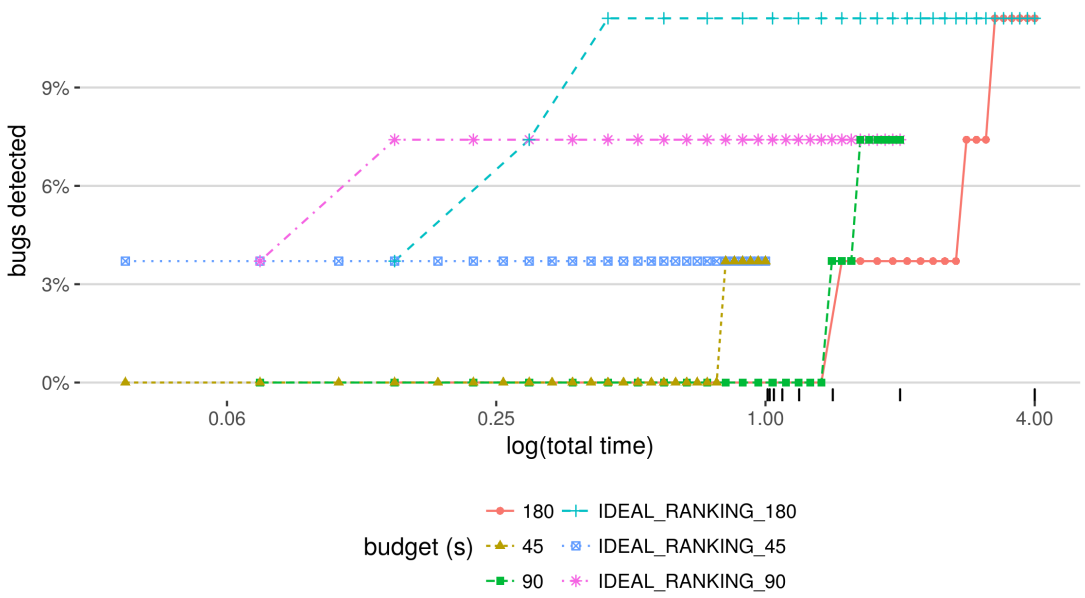

Figure A.4: *Total generation cost for bug detection.* Performance of COVERAGE_DENSITY_COMBO ranking at different budgets for the Time project and Randoop.

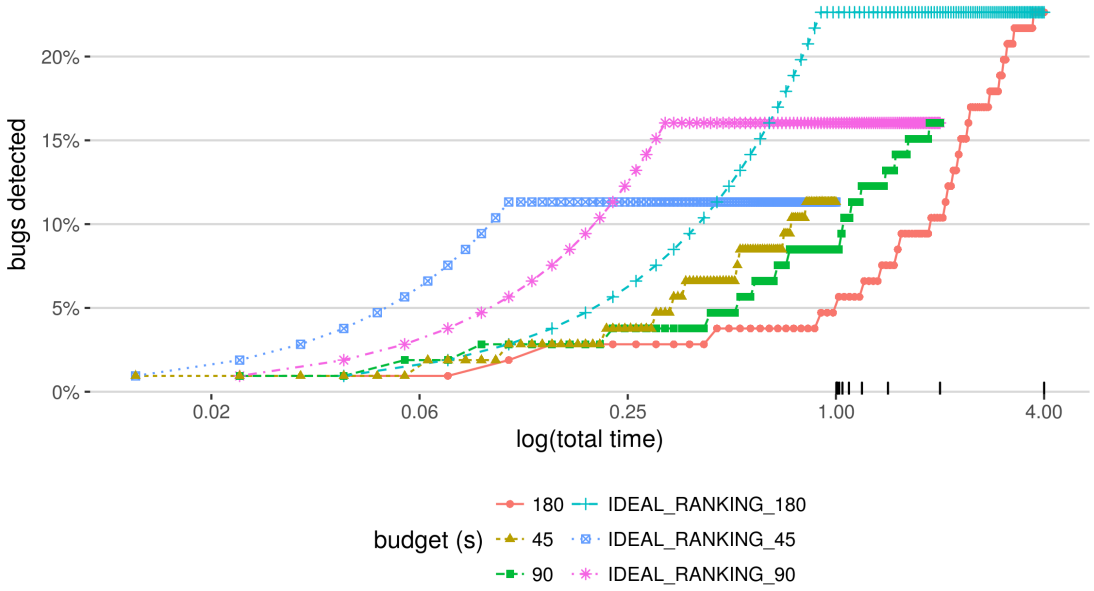

Figure A.5: *Total generation cost for bug detection.* Performance of COVERAGE_DENSITY_COMBO ranking at different budgets for the Math project and Randoop.

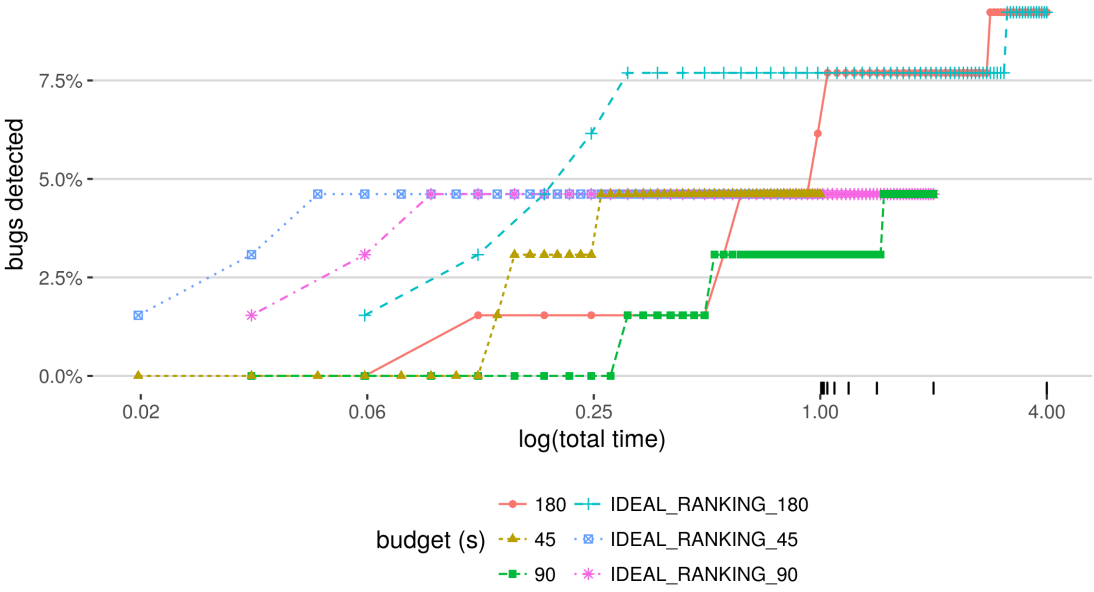

Figure A.6: *Total generation cost for bug detection.* Performance of COVERAGE_DENSITY_COMBO ranking at different budgets for the Lang project and Randoop.

A.3 Plots for COVERAGE_DENSITY_COMBO at different budgets

Here, additional plots of the best ranking method COVERAGE_DENSITY_COMBO at different budgets are included. The purpose is to show that smaller budgets don't affect bug detection strongly. The difference compared to the "cost plots" presented in the results chapter is that the x-axis is not log-transformed. This makes it easier to see that the shape of the curves is almost identical. There are times when bug detection is the same at different budgets. This is the case for the Time project and EvoSuite in fig. A.7. The biggest difference in bug detection is in fig. A.11, where there is a relative difference of about one third. What is important though is that in all cases the shapes of the curves are very similar. There are also no crossings. This is an indication that the rankings are stable even at lower budgets.

A.3.1 EvoSuite

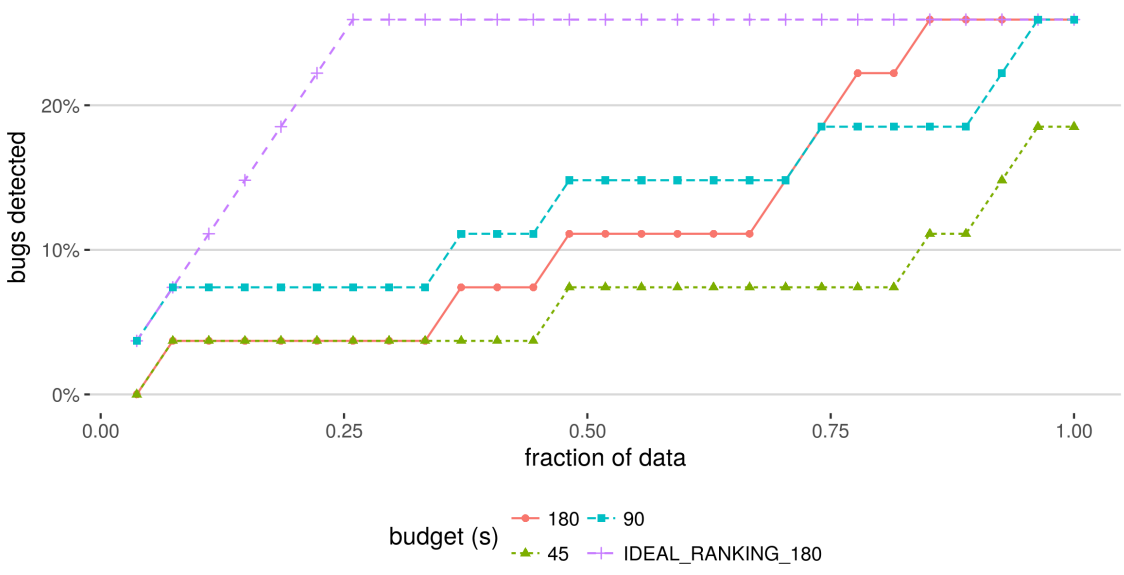

Figure A.7: *Bug detection as a function of ranked data.* Performance of COVERAGE_DENSITY_COMBO ranking at different budgets for the Time project and EvoSuite. Budgets 90 and 180 achieve the same bug detection.

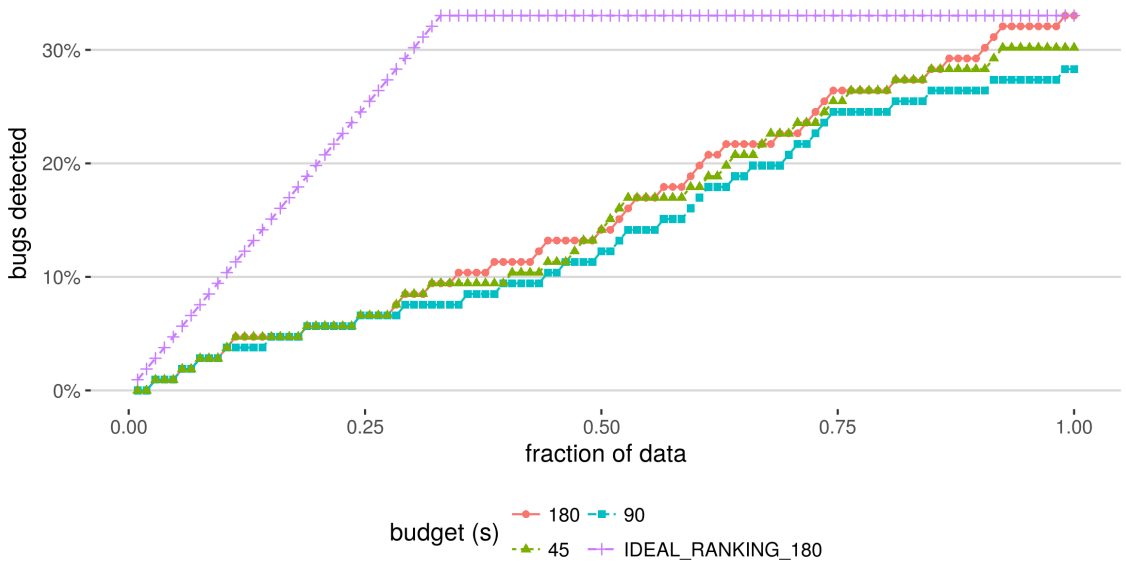

Figure A.8: *Bug detection as a function of ranked data.* Performance of COVERAGE_DENSITY_COMBO ranking at different budgets for the Lang project and EvoSuite. There is no significant difference in bug detection.

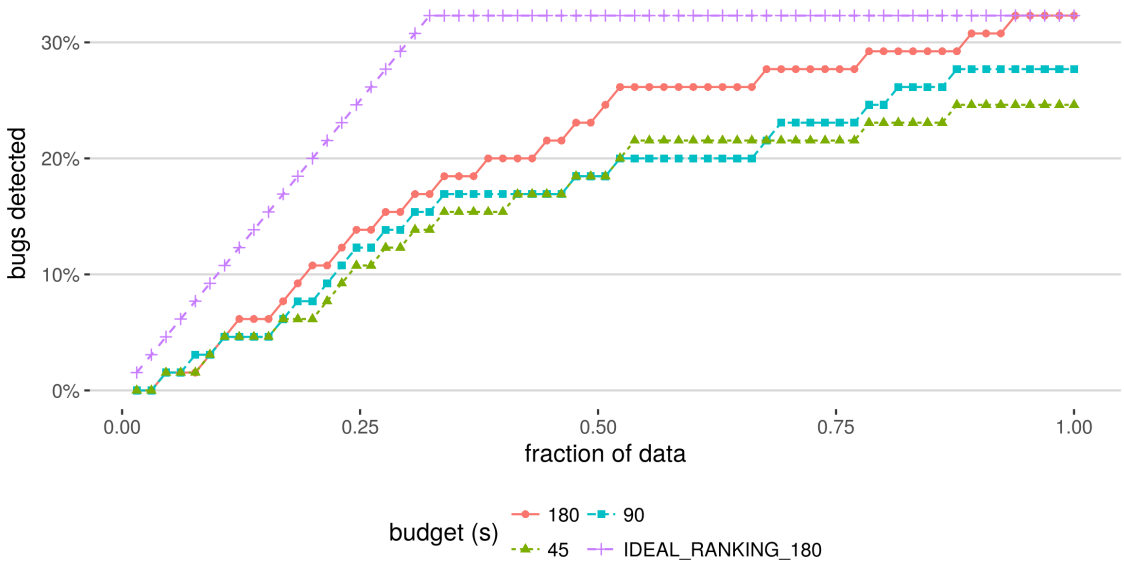

Figure A.9: *Bug detection as a function of ranked data.* Performance of COVERAGE_DENSITY_COMBO ranking at different budgets for the Lang project and EvoSuite. The biggest budget results in best bug detection. The curve remains on top of the other curves of lower budgets throughout the whole time as expected.

A.3.2 Randoop

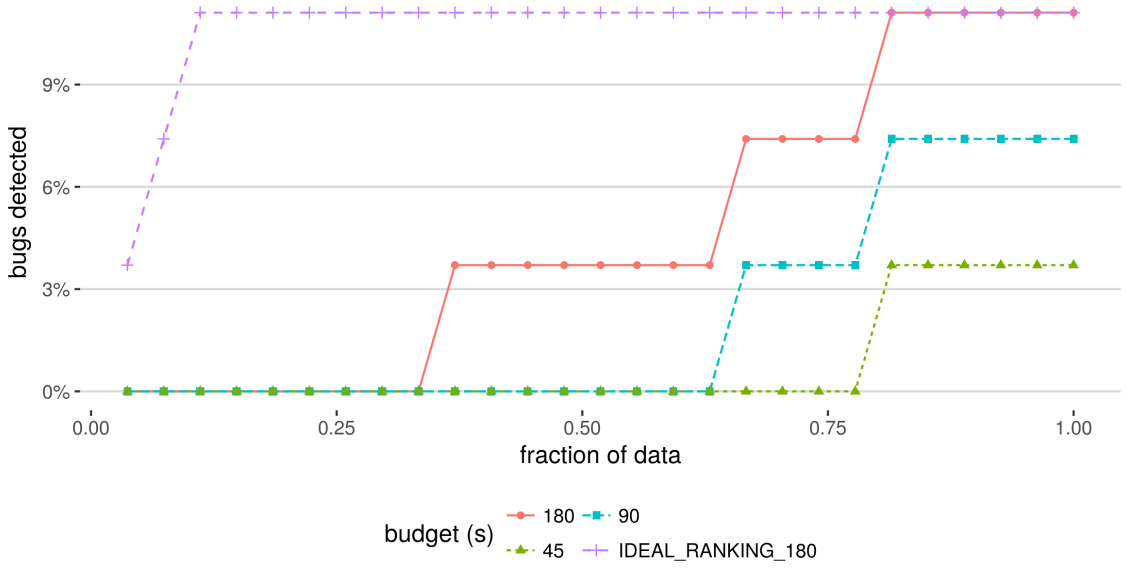

Figure A.10: *Bug detection as a function of ranked data.* Performance of COVERAGE_DENSITY_COMBO ranking at different budgets for the Time project and Randoop. There is a big relative difference in bug detection. However, this in absolute terms this is tiny (one bug difference).

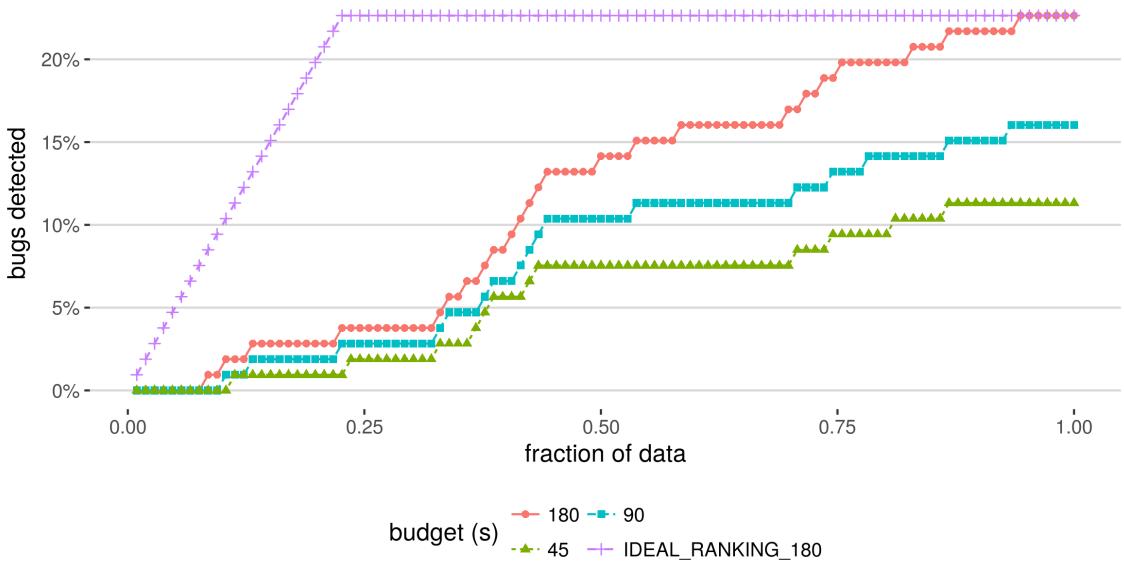

Figure A.11: *Bug detection as a function of ranked data.* Performance of COVERAGE_DENSITY_COMBO ranking at different budgets for the Math project and Randoop. This probably shows the largest difference in bug detection. However, all the curves have similar shapes confirming the expectation that the ranking is stable.

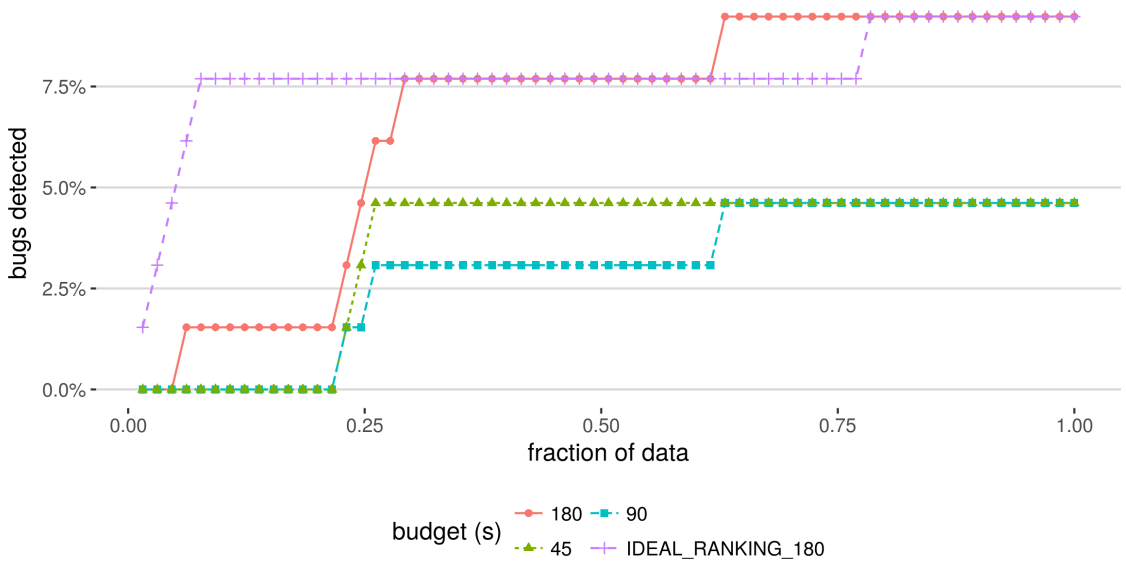

Figure A.12: *Bug detection as a function of ranked data.* Performance of COVERAGE_DENSITY_COMBO ranking at different budgets for the Lang project and Randoop.

A.4 Branch coverage ranking for Randoop

The AUC plots for the goal of branch coverage and testing Randoop are presented here. We can see here similar behavior of ranking methods like for EvoSuite. The methods BUG, LOC_DESC and COVERAGE_BRANCH follow along a similar path. The COVERAGE_DENSITY_COMBO method is the worst performer here, while COVERAGE_PREDICTION method is in-between the extremes. The COVERAGE_DENSITY_COMBO curve is smoother than that of its component COVERAGE_PREDICTION. A plausible explanation for this is that the CDC method is more influence by LOC. It is not clear why the BUG method follows the COVERAGE_PREDICTION method so closely in fig. A.15, because the methods are unrelated.

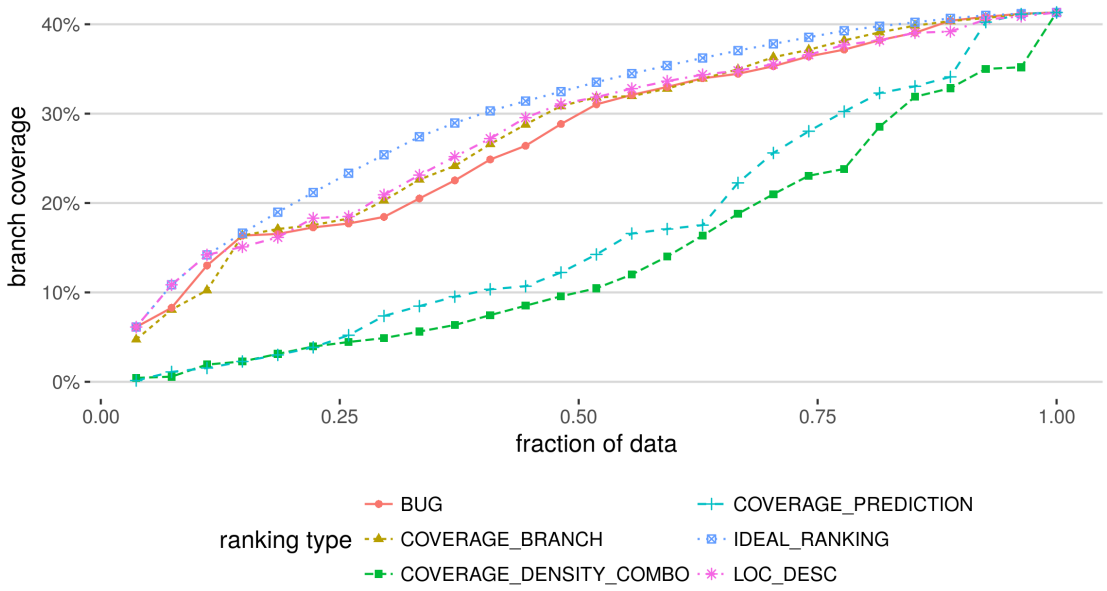

Figure A.13: *Branch coverage as a function of ranked data.* Branch coverage ranking for the project Time and Randoop.

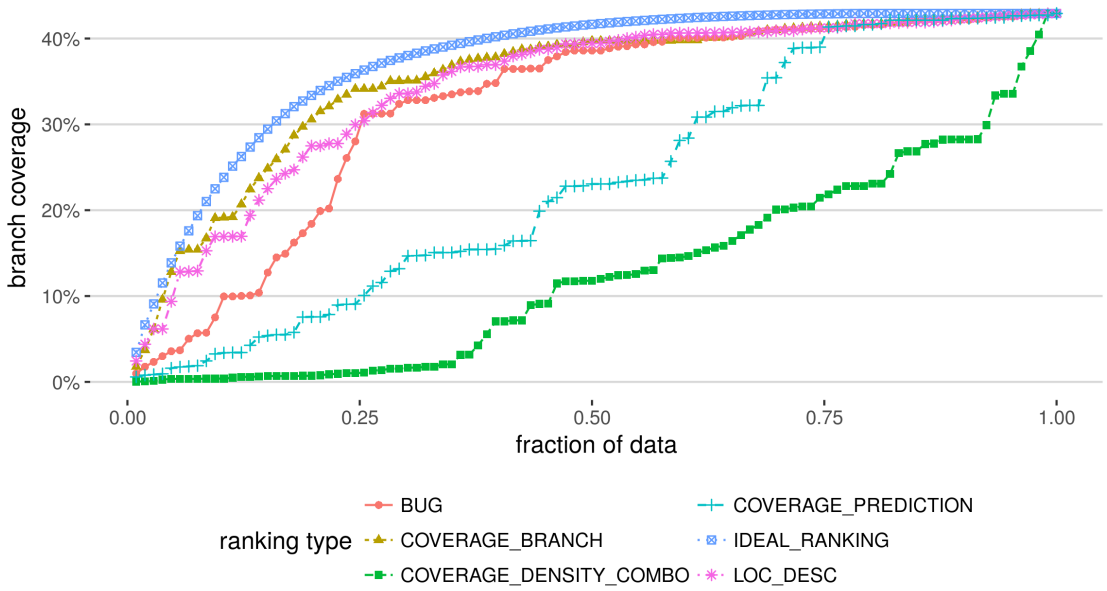

Figure A.14: *Branch coverage as a function of ranked data.* Branch coverage ranking for the project Math and Randoop.

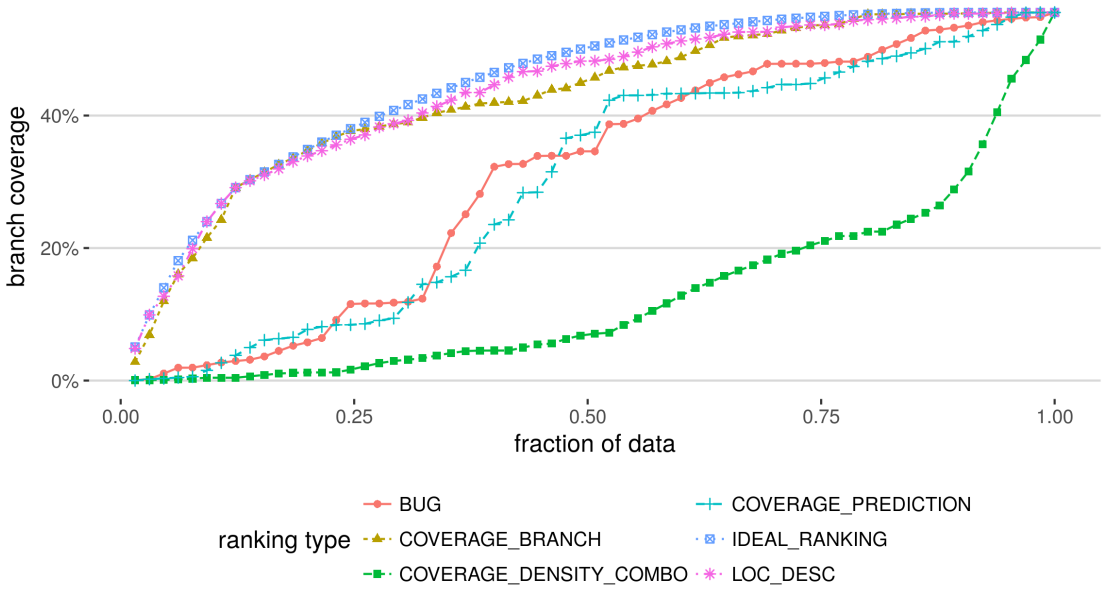

Figure A.15: *Branch coverage as a function of ranked data.* Branch coverage ranking for the project Lang and Randoop.