EXPLOITING USER FEEDBACK TO FACILITATE
OBSERVATION-BASED TESTING

By
VINAY JOSEPH AUGUSTINE

Submitted in partial fulfillment of the requirements
For the degree of Doctor of Philosophy

Dissertation Advisor: Dr. H. Andy Podgurski

Department of Electrical Engineering and Computer Science
CASE WESTERN RESERVE UNIVERSITY

January, 2010
We hereby approve the thesis/dissertation of

______________________________________________________

Vinay J. Augustine

candidate for the Ph.D. degree *

(signed) ____________________________________________

H. Andy Podgurski

(Chair of the committee)

__________________________________________________

Gultekin Ozsoyglu

__________________________________________________

Jiong Yang

__________________________________________________

Jiayang Sun

__________________________________________________

__________________________________________________

(date) ________________

13 July 2009

*We also certify that written approval has been obtained for any proprietary material contained therein.
This work is dedicated to my parents, Dr’s Jose and Loretta Augustine, for never giving up on me; and my wife, Angela Augustine, for keeping me sane.
Contents

1 Introduction 1

2 Related Work 8
   2.1 Classification ........................................... 8
   2.2 Test Selection and Prioritization ...................... 9
   2.3 Reliability Estimation ................................. 10
   2.4 Collecting data from users ............................ 10
   2.5 Collaborative Filtering ................................ 11
   2.6 Active learning ........................................... 13
   2.7 Fault localization ....................................... 14

3 A Case Study of Mislabling by Users 17
4 Techniques

4.1 Data Collection and Analysis .................................................. 25
4.2 Cluster Filtering with One-Per-Cluster Sampling ......................... 26
4.3 Review All FAILUREs ............................................................... 26
  4.3.1 RAF+ ................................................................. 27
4.4 Corroboration-based Filtering .................................................. 27
  4.4.1 Filtering Rules ........................................................... 28
  4.4.2 Probabilistic Analysis .................................................... 29
4.5 Review-All-failures and $k$-Nearest Neighbors .......................... 31
4.6 Round-robin Cluster Filtering with $k$-NN Ranking ....................... 33
  4.6.1 $k$-NN Ranking .......................................................... 34
4.7 Assumptions ........................................................................... 34

5 Experimental Methodology ......................................................... 35

5.1 Subject Programs ................................................................. 36
5.2 Data Analysis .......................................................................... 38
5.3 Study of Corroboration-based Filtering ..................................... 39
5.4 Study of RAF+$k$NN .............................................................. 40
5.5 Study of Relative Costs of CBF, RAF, and RAF+$k$NN ................. 41
5.6 Study of RRCS with $k$-NN Ranking ......................................... 41

6 Evaluation of Corroboration-Based Filtering ................................ 43

6.1 Failures Discovered ............................................................... 43
6.2 Defects Discovered ............................................................... 51
6.3 Executions Reviewed .............................................................. 52
6.4 CBF with $T = 0$ ..................................................................... 58
  6.4.1 Failures Discovered ........................................................ 58
  6.4.2 Defects Discovered ........................................................ 62
6.4.3 Executions Reviewed ........................................... 63

7 Evaluation of Review-All-FAILUREs and $k$-Nearest Neighbors 68
  7.1 Comparing CBF and RAF+$k$NN ................................. 68
  7.2 Failures Discovered .............................................. 69
  7.3 Defects Discovered ............................................... 72
  7.4 Executions Reviewed ............................................. 74

8 Analysis of Relative Costs of CBF, RAF+$k$NN, and RAF 81
  8.1 Effects of Differential Mislabelling on Relative Cost .............. 87

9 Evaluation of Round-robin Cluster Sampling with $k$-NN Ranking 88
  9.1 Failures Discovered .............................................. 89
    9.1.1 $k$-NN Ranking ............................................ 94
  9.2 Defects Discovered ............................................... 95

10 Conclusions and Future Work .................................... 104
  10.1 Future Work ................................................... 109

Bibliography .......................................................... 110
### List of Tables

3.1 Compiler Results ................................................. 18
3.2 Chapters from JLS ............................................... 20
5.1 Subject Data ..................................................... 36
5.2 Experimental Data ............................................... 38
6.1 Mean failures discovered with OPC .............................. 49
6.2 Percentage mean defects discovered with $p_{ml} = 0.3$ ........ 51
6.3 Percent mean defects discovered with OPC ....................... 52
6.4 % Increase in $R$ and $F_d$ from RAF to CBF ($p_{ml} = 0.3$ and $T = 1$) ... 57
6.5 Percentage mean defects discovered by CBF with $p_{ml} = 0.3$ .... 62
7.1 $R/F_d$ for CBF with $p_{mf} = 0.3$ and $T = 1$ .................... 69
9.1 Maximum Number of Properly Labeled Failures for differing $p_{mf}$ Values 88
9.2 Maximum Number of Executions Reviewed with RRCS for differing $p_{ms}$ Values ............................... 89
9.3 Mean Executions Reviewed vs. Failure Discovered for $k = \{0, 1, 3\}$ . 93
List of Figures

1.1 Mozilla Firefox Error Reporting Dialog ........................................... 2

3.1 Sample Negative Test Case from Jacks Test Suite (16.2.8-unassigned-1) 19
3.2 Histogram of Number of Correct Answers for each User ................... 20
3.3 Histogram of Number of Correct Answers by JLS Test Case Answer . 21
3.4 Histogram of Number of Correct Answers by JLS Chapter ................. 22

6.1 % Mean Failures Discovered vs. $p_{ml}$ for differing $T$ values .......... 45
6.2 % Failures Discovered for Different T Values ............................... 47
6.3 % Mean Failures Discovered vs. $p_{ml}$ for CBF, RAF+, RAF .......... 50
6.4 % Mean Executions Reviewed vs. $p_{ml}$ for CBF for differing $T$ values 53
6.5 Executions Reviewed for differing $T$ values .................................. 55
6.6 % Mean Executions Reviewed vs $p_{ml}$ for CBF and RAF ............... 56
6.7 Executions reviewed per failure discovered vs $p_{ml}$ for CBF and RAF . 59
6.8 % Mean Failures Discovered vs $p_{ml}$ for CBF0,CBF1, and RAF ...... 61
6.9 % Mean Executions Reviewed vs $p_{ml}$ for CBF0,CBF1, and RAF ... 65
6.10 Executions reviewed per failure discovered vs $p_{ml}$ for CBF0, CBF1, and RAF ............................................................... 67

7.1 % Mean Failures Discovered vs. $p_{mf}$ ........................................... 70
7.2 % Mean Defects Discovered vs. $p_{mf}$ ........................................... 73
7.3 % Mean Executions Reviewed vs. $p_{mf}$ ........................................... 76
7.4 Executions reviewed / Failure Discovered vs. $p_{mf}$ .......................... 78
8.1 GCC: Total Cost vs. Multiplier .................................................. 82
8.2 Javac: Total Cost vs. Multiplier .................................................. 83
8.3 JTidy: Total Cost vs. Multiplier ................................................... 84
8.4 ROME-F 5%: Total Cost vs. Multiplier ...................................... 85
8.5 ROME-L 5%: Total Cost vs. Multiplier ...................................... 85
8.6 Xerces 5%: Total Cost vs. Multiplier ......................................... 86
9.1 Mean Executions Reviewed per Failure Discovered for GCC ............. 90
9.2 Mean Executions Reviewed vs. Failure Discovered for Javac .......... 91
9.3 Mean Executions Reviewed vs. Failure Discovered for JTidy .......... 92
9.4 Mean Executions Reviewed vs. Failure Discovered for ROME ......... 92
9.5 Mean Executions Reviewed vs. Failure Discovered for Xerces 5% .... 94
9.6 Mean Executions Reviewed vs. Defects Discovered for GCC ........... 95
9.7 Mean Executions Reviewed vs. Defects Discovered for GCC with $k = \{0, 1, 3\}$ .......................................................... 96
9.8 Mean Executions Reviewed vs. Defects Discovered for Javac ........ 97
9.9 Mean Executions Reviewed vs. Defects Discovered for Javac $k = \{0, 1, 3\}$ 98
9.10 Mean Executions Reviewed vs. Defects Discovered for JTidy ........ 98
9.11 Mean Executions Reviewed vs. Defects Discovered for JTidy $k = \{0, 1, 3\}$ 99
9.12 Mean Executions Reviewed vs. Defects Discovered for ROME ....... 100
9.13 Mean Executions Reviewed vs. Defects Discovered for ROME $k = \{0, 1, 3\}$ 101
9.14 Mean Executions Reviewed vs. Defects Discovered for Xerces 5% .... 102
9.15 Mean Executions Reviewed vs. Defects Discovered for Xerces 5% $k = \{0, 1, 3\}$ 103
I would like to acknowledge the tireless work by my adviser, Dr. Andy Podgurski, in working with me. Countless conversations with him have lead to this research. My abilities as a researcher are, in large, part, due to his instruction and guidance.

I would also like to thank my committee: Dr. Gultekin Ozsoyoglu, Dr. Jiong Yang, and Dr. Jiayang Sun. You’ve all helped me refine these techniques, and the classes I’ve taken with each of you have given me many new directions to pursue.

Patrick Francis, Will Wong, and Eldan Goldenberg deserve special mention for reviewing multiple drafts and suggesting myriad improvements.

Finally, I cannot thank my family enough for the love and support they’ve given me. None of this would have been possible without my parents, Dr’s. Jose and Loretta Augustine, my sister, Sapna, and my wife, Angela.
EXPLOITING USER FEEDBACK TO FACILITATE OBSERVATION-BASED TESTING

Abstract

by

VINAY JOSEPH AUGUSTINE

Recent progress in the use of data mining and statistical techniques to automatically classify related software failures and to localize defects suggests that if appropriate information is collected about executions of deployed software, then such techniques can assist developers in prioritizing action on soft failures reported by users and in diagnosing their causes. Users, however, are not reliable judges of correct software behavior: they may overlook real failures, neglect to report failures they do observe, or report spurious failures. Instead, I propose to employ users as independent checks on each other. Previous work demonstrated that executions with similar execution profiles often represent similar program behavior. By grouping similar executions together, developers can use user-submitted labels to corroborate each other: similar executions with the same label represent consensus, and similar executions with differing labels represent suspicious or confusing behavior.

An empirical evaluation of two proposed techniques, Corroboration-based Filtering, Review-All-FAILUREs plus k-Nearest Neighbors, indicates that they discover significantly more failures and defects than the naive review-all-FAILUREs strategy. A third technique, round-robin cluster sampling, discovers failures and defects more quickly than RAF.
Chapter 1

Introduction

Both large and small software projects are increasingly relying on beta testing to discover defects before an official release. This is largely due to the ease with which developers and users can communicate through the use of the internet. In particular, projects will give pre-release versions of their software to users who are interested in testing new features or verifying that older defects are fixed.

Feedback from users is managed through a variety of interfaces: projects will set up mailing lists, newsgroups, or online forums, to gather comments from users. More complicated feedback is gathered through the use of issue tracking systems, which, in addition to descriptive comments, also collect severity ratings, and specific features affected.

If possible, projects will collect feedback from within the application itself. Many applications are often setup to submit crash data, along with a user-supplied description of what the user was doing when the application crashed. This makes it more likely that developers receive crash data. Failure reporting promises to become even more useful with the advent of recent research that applies data mining and statistical techniques to execution data in order to help developers automatically categorize software failures [10, 30, 38] and localize the defects (faults) that caused them [3, 29].
Figure 1.1: Mozilla Firefox Error Reporting Dialog
Unfortunately, application crash logs only record a subset of failures that may occur in the field. Issues involving incorrect outputs and unusual or unexpected program behavior are examples of defects that do not always result in program crashes. In order to discover these soft failures, it is desirable to equip software products with a reporting feature that allows users to submit reports about both hard failures (resulting in a crash) and soft failures. For example, while the Firefox web browser can be configured to automatically send crash data to developers, it can also collect user-submitted feedback regarding individual websites. Figure 1.1 shows the reporting dialog available from the Firefox help menu. Here, users can submit a description of the perceived error and give the URL where the error was encountered. In order to facilitate the discovery of these soft failures, it is desirable to equip software products with a similar reporting feature that, along with user reports about hard and soft failures, sends execution data that will aid developers in confirming and categorizing failures and diagnosing their causes.

One obstacle to exploiting these user reports is that they may be noisy. Beta testers are not completely reliable judges of correct software behavior: they may overlook real failures, neglect to report failures they do observe, or report spurious failures [19]. Because of this, developers cannot rely on their feedback with the same confidence that they rely on feedback from developers. Instead, I propose to employ users as independent checks on each other. Previous work in cluster filtering [7, 8, 27, 28] indicates that using distance-based techniques to group similar executions is effective.

This dissertation presents a collection of techniques that allow developers to review “suspicious” executions. Ideally, developers reviewing suspicious executions should review mostly failures and relatively few successes, regardless of whether users correctly labeled them or not.

These techniques can be used by developers to prioritize which submitted execu-
tions get reviewed based on project resources and needs. All three techniques group executions via some distance-based techniques such as \( k \)-nearest neighbors or \( k \)-means clustering. Once the executions are grouped, developers can select specific groups for prioritization. Cluster filtering [7] defined “suspicious” executions as having profiles that were dissimilar from other profiles collected. This work expands on that measure by assuming that similar executions should have similar labels. Developers can therefore ignore areas of the profile space believed to contain only successful executions (based, in part, on developer review) and only review those areas that contain likely failures (executions labeled FAILURE by users) or suspicious executions (executions that are near each other but have differing labels). Developers should be more likely to review executions recommended by one of these techniques if they are reviewing executions that lead to the discovery of failures or defects.

Other user-powered classification systems (described in Section 2.4) assume that users are generally correct. Any errors by individual users are assumed to be rare and will therefore be de-emphasized. In general, this holds true: the types of data collected usually involve user opinion where there is no incorrect answer. The problem presented here, however, is different: users can make mistakes that can lead developers to review successful executions that do not help them find defects.\(^1\)

There are two scenarios to consider. In the first scenario, developers are faced with expensive failures and defects. In this context, an expensive defect is one where the resulting failures may result in significant harm to the user. This harm may stem from lost or corrupted data or failure of some mission-critical system. Equally important, it should measure the economic harm to an organization due to lost revenue and reputation. In the second scenario, developers either do not expect catastrophic failures, or do not have the resources to review a large number of executions. Because

\(^1\)It should be noted that application behavior that results in a large number of erroneous labels may indicate a poorly designed feature — a possible defect. How this threshold is selected is not considered here.
of this, they focus exclusively on reported failures.

All of the techniques are compared against a basic Review-All-FAILUREs (RAF) technique. With RAF, developers simply review all executions labeled FAILURE. Assuming users do not make too many mistakes, RAF is likely to reveal a high number of actual failures and defects.

The first technique, Corroboration-based Filtering (CBF), defines three classes of suspicious executions. In order to do this, CBF uses a clustering algorithm (such as $k$-means) to group similar executions together. CBF filters the set of executions by looking at what kinds of executions are in the same cluster. First, CBF defines any execution that users have labeled FAILURE to be suspicious. Developers are directed to review all of these executions. Next, CBF defines any execution that is in the same cluster as a confirmed failure to be suspicious. If an execution is in the same cluster as a failure, then there is evidence that that cluster represents some defect. If, however, there are no confirmed failures in a cluster, then it is likely that the cluster contains only successful executions. Developers can ignore these clusters since they represent correct application behavior. Finally, CBF directs developers to review unusual executions. An unusual execution is an execution in a very small cluster (controlled by a threshold, $T$). Executions in small clusters represent unusual behavior in the application — even if labeled SUCCESS, developers may want to review these executions to verify they have worked correctly. Because these clusters only contain a few executions, it is possible that there is not enough evidence to dismiss the cluster — for instance, if a singleton cluster is marked SUCCESS, there is no way to know if this was a mistake or a user evaluating the output.

The second technique, Review-All-FAILUREs and $k$ Nearest Neighbors (RAF+$k$NN), defines two classes of suspicious executions. Like CBF and RAF, RAF+$k$NN also begins with developers reviewing all executions labeled FAILURE. Then, like CBF, RAF+$k$NN defines the second class of suspicious executions to be those in the neigh-
borhood of a confirmed failure. However, instead of using clustering, RAF+$k$NN uses a nearest-neighbor algorithm. Developers review the $k$ nearest neighbors of each confirmed failure. By evaluating the $k$ nearest neighbors, developers have greater control over how many executions they will evaluate. In the same vein, RAF+$k$NN does not direct developers to review “unusual” executions.

These two techniques, corroboration-based filtering and review-all-FAILUREs and $k$ nearest neighbors, attempt to discover as many failures and defects as possible. They are meant to be used in situations where the cost of missing both failures and defects is high.

The third technique, Round-Robin Cluster Sampling with $k$-NN Ranking (RRCS) takes a different approach from both CBF and RAF+$k$NN. With the first two techniques both direct developers to review both FAILURE and SUCCESS labeled executions, because developers cannot risk missing a defect. With RRCS, developers only consider executions labeled FAILURE by users. Because all of these executions are considered suspicious, RRCS basically attempts to direct developers to the executions most likely to be failures. RRCS begins by clustering all of the FAILURE-labeled executions. Then, developers select one execution from each cluster for review. This process is repeated, either until all FAILURE-labeled executions are exhausted, or developers have found a requisite number of failures or defects. By reviewing executions in this round-robin fashion, developers can explore the entire profile space in a systematic fashion. Assuming that failures are clustered according to their root cause, developers will discover more defects relatively quickly. RRCS also has an extension, $k$-NN ranking, which helps to prioritize the executions within a particular cluster. After $k$ executions from each cluster have been reviewed, the remaining executions are prioritized based on how many confirmed failures exist in the $k$ nearest neighborhood.

Empirical studies are presented which evaluate each technique to determine if it
discovers more actual failures and defects, and if it requires more work (in terms of
developers reviewing executions) than the naive technique of reviewing all FAILURE-
labeled executions. An analysis of the relative costs of CBF, RAF+\(k\)NN, and RAF
is given, allowing organizations to examine the tradeoff between developer time and
allowing defects to remain undiscovered.
Chapter 2

Related Work

2.1 Classification

Podgurski et al. [38] presented a technique that applies cluster analysis, supervised learning, and multivariate visualization techniques to execution profiles in order to classify reported software failures according to their underlying defects. Francis et al. [10] presented two tree-based techniques for refining an initial classification of failures. Neither of these techniques address the possibility of erroneous failure reports by users.

Kailing et al. [25] design a way to cluster objects represented by multiple feature sets. This technique extends density-based clustering (DBSCAN) to handle multiple feature sets. The authors introduce two new features to DBSCAN: (1) union, which places objects in the same cluster as long as they are similar in at least one feature space and (2) intersection, which places objects in the same cluster only if they are similar in all feature spaces. Unions are stated to be useful for dealing with sparse data, and intersections are stated to be useful when the feature sets do not provide complete descriptions of the data points. In an empirical study, unions were useful for clustering proteins by amino-acid data as well as textual descriptions. Intersections were useful for classifying images by color histograms and text representations. In
this case, neither feature set alone was sufficient for describing the data; together, however, they produced useful clusters. These techniques differ from this work in that they do not explicitly deal with errors in the data, instead, it seeks to combine disparate representations of a dataset without having to construct a special distance metric.

2.2 Test Selection and Prioritization

Cluster filtering of execution profiles and several variants are presented and evaluated empirically by Dickinson et al. [7]. Cluster filtering attempts to isolate failures in an execution set into a number of small clusters. Sampling strategies involving these clusters then select a number of executions to evaluate. Leon and Podgurski [27] presented an empirical comparison of test suite minimization, prioritization by additional coverage, cluster filtering with one-per-cluster sampling, and failure pursuit sampling. Leon et al. [28] presented an empirical evaluation of test case filtering techniques used in conjunction with profiles of information flow and dynamic slices. These techniques do not rely on user labeling; instead the sampled executions are reviewed by developers to determine their success/failure status. One of the techniques, cluster filtering with one-per-cluster sampling (OPC) is compared to the methods presented here. OPC is described in Section 4.2.

Ramanathan et al. [41] present PHALANX, a prioritization framework for regression test suites. PHALANX represents test cases using a user-selected profiler, such as function-call sequences. A dissimilarity graph is created in which undirected edges are weighted using a cost function based on the Levenstein edit distance. Test cases are prioritized by finding the most dissimilar cases. Two techniques, a greedy approach and an approach based on a spectral ordering of the graph, are compared. PHALANX differs from the work presented here in that it is intended for use with
regression test suites; as such, it does not take into account user feedback and errors in reporting.

### 2.3 Reliability Estimation

Podgurski et al. [37] investigated clustering and stratified sampling of execution profiles for improving the accuracy of software reliability estimation. These techniques do not require feedback from users, instead profile data is automatically collected from users via capture/replay and then analyzed by developers. In addition, the work presented here does not attempt to measure reliability; instead it focuses on the discovery of failures and defects.

Jalote et al. [20] study how to measure the reliability of both mass-market and server products. Their approach includes a user-centric classification of defects, operational frequency of those defect types, and the usage time of the product. Failure events are collected through a combination of user polls and automatic event collection. Their approach focuses on measuring reliability and does not take into account errors in user-submitted reports.

### 2.4 Collecting data from users

Ha et al. [17] presented Clarify, a system that classifies program behavior in order to improve error reports. Clarify classifies failing execution profiles using a machine learning classifier. Developers train Clarify using execution profiles received from users. The developers then write improved error messages for each error class using features identified by the classifier. Ha et al. also present a new form of profiling called call-tree profiling that is more accurate than previous forms of profiling such as function call and control flow. Clarify is primarily intended to allow developers to write improved error messages based on input from users, while our techniques are
aimed at discovering software defects.

Michail and Xie [33] described an interactive tool called *Stabilizer* for helping users of a GUI application avoid buggy program behavior, which is similar to our approach in several respects. *Stabilizer* permits users to report failures and records partial execution histories associated with them. A weighted nearest-neighbor learner is applied to the current execution history and recorded ones to determine if the current history is similar enough to past failure-inducing histories to justify warning the user so they can take alternative actions. The user can indicate that a failure warning was mistaken and thereby supply negative training instances. Labeled execution histories from different users can be employed with the learner. Michail and Xie’s work differs from ours in the following ways: the application is bug avoidance rather than validating software behavior; developers don’t review suspect executions; *Stabilizer* does not employ profiles of entire executions; it does not necessarily treat ”isolated” executions as suspicious; and it was not evaluated for robustness to user mislabeling.

Raddick et al. [40] developed Galaxy Zoo, a website devoted to the manual classification of galaxies by non-experts. Users are given a brief tutorial with examples showing how to classify commonly seen galaxy features and what to do if the user cannot decide. Galaxy Zoo uses numerous classifications per galaxy to come to a consensus on the galaxy’s actual classification. This differs from the techniques presented here, in that, instead of collecting multiple classifications per execution, we group similar executions together and use that to look for consensus.

### 2.5 Collaborative Filtering

Collaborative filtering relies on users to provide assessments of the usefulness of information. Collaborative filtering is used to provide users with product recommendations and relevant news or blog articles. Collaborative filtering works by comparing users’
usage patterns and then making recommendations to them based on similar users’ usage patterns.

Collaborative filtering can be divided into two areas:

1. Item-based filtering: In item-based collaborative filtering, users rate particular objects (examples include purchases, movie rentals, or restaurants). Usage and rating patterns of these objects are then analyzed. In item-based collaborative filtering, users rank particular items that may be recommended to other users.

2. Content-based filtering: In content-based collaborative filtering, users rank particular objects, but comparisons may include classes of objects. This classification is computed using a description of the object. Item recommendations are then generated based on classes that users share.

The work presented here essentially applies collaborative filtering techniques to the discovery of software defects. User labels are used to find clusters of executions that may contain failures. In general, however, collaborative filtering does not take into account errors by users (because it is typically used to correlate users’ opinions, errors are hard to define).

In software engineering, collaborative filtering has been used to facilitate sharing of knowledge between developers and to produce more accurate estimates for project development.

Ohira et al. [35] used collaborative filtering and social network analysis to facilitate knowledge sharing among developers of open source software projects. Ohira et al. created tools that allowed developers to find others who were working on similar projects, in the hopes that they might share solutions to problems. This work did not focus on the software development process; instead it created tools for fostering developer community.

McC Carey et al. [32] created a system called RASCAL, that recommends software
library components to software developers based on the tasks that they are currently solving. *RASCAL* uses collaborative filtering to identify existing source code that is similar to code that is currently being worked on. If developers working on similar tasks (based on source code similarity) used components of a reusable library, then these components will be automatically recommended. Here, the authors focused on the knowledge dissemination during development, instead of the discovery of failures or defects.

Mitani et al. [34] used collaborative filtering on benchmarks for completed software projects in order to create more accurate estimates for currently operating projects. Collaborative filtering was used to identify similar projects so that comparisons could be made between those projects and a currently executing project. The authors focused on creating accurate measurements of reliability instead of the discovery of software defects.

Cleland-Huang and Mobasher [4] proposed using collaborative filtering as part of a requirements collection system that scales up to “Ultra-Large-Scale” projects. Ultra-Large-Scale projects are projects with thousands or hundreds of thousands of stakeholders; something software development models were not designed to deal with. By using collaborative filtering in conjunction with other data analysis techniques, a prioritized list of requirements is created that should be more representative of the needs of the various stakeholders.

### 2.6 Active learning

In active learning, users typically have a collection of *unlabeled* data, which they want to classify without having to review each data point. To this end, various data analysis techniques are used to classify sets of executions based on a sample of manual reviews. This differs from the techniques presented here, in that developers have semi-reliable
labels that they can use to guide their search.

Pelleg and Moore [36] described an active learning technique that employs users to identify interesting anomalies. The learning technique presented takes unlabeled data, classifies the data via semi-supervised learning, then asks experts to classify the “strangest” data, and repeats. This technique is applied to the classification of images for astrophysics research. This research targeted astrophysics, not software testing, and does not focus on discovering a particular category (such as “defect”). In addition, this algorithm is supervised, and is refined by experts.

He and Carbonell [18] described a nearest-neighbor based active learning technique that works for multiple classes. As with [36], focuses on “rare” categories. The authors uses an unsupervised learning technique to discover categories in unlabeled data. This work does not assume user labeling; instead it is evaluated for estimation errors for a prior categorization.

Dasgupta and Hsu [6] described a sampling method that uses hierarchical clustering to avoid sampling bias. As with other active learning techniques, hierarchical sampling does consider user labels; it begins with unlabeled data and then uses sampling to infer the labels of the unsampled data.

2.7 Fault localization

Jones et al. [23] present parallel debugging, a method that allows developers to distinguish between multiple defects exposed by an execution failure. Parallel debugging involves the generation of fault focusing clusters. Each fault-focused cluster represents a specialized test suite that can be debugged by a different developer. This clustering is produced through agglomerative hierarchical clustering on branch profiles. The number of clusters is selected by applying the Tarantula [22] fault-localization technique to determine whether two clusters contain the same set of suspicious statements.
This work differs from the work presented here, in that parallel debugging was not evaluated in the context of beta testing. However, because parallel debugging requires both failing and successful executions, some of the methods presented here (CBF and RAF+NN) can be used to produce a labeled set of executions usable for parallel debugging. To analyze parallel debugging, the authors present an expense metric based on the number of statements that must be analyzed by developers in order to fix the defect. This differs from the cost metric discussed in Chapter 8, which analyzes the cost to the development organization in terms of defects missed and work required.

Liblit et al. [29] designed an approach to collecting information about program executions from a user community and automatically analyzing the information to help in isolating bugs. Zheng et al. [49] used this sampling framework to find defective code via a utility function designed to pinpoint useful sampling points. This work focuses on defects that cause crashes and does not involve classification of failures. In [50], this work was expanded to allow identification of multiple defects through the application of bi-clustering. Both of these techniques are used to debug hard defects, though it may be possible to debug soft defects with user-reported failures.

Bodik et al. [1] combine statistical anomaly detection with visualization techniques in order to identify and localize faults for a web application. Naive Bayes classification and chi square were used to identify unusual changes in the hit frequencies of the top forty pages on a site. A heat map was used to visualize these hit counts per minute. Site operators are notified when traffic anomalies detected; these anomalies can be queried to see which pages experienced the greatest change in traffic. This work relies on changes in user behavior to discover failures. Localization is provided by examining the most suspicious pages during an anomalous event. This work, on the other hand, requires explicit or implicit labeling from users and combines that with a distance-based analysis of collected profiles.
Chen et al. [3] present a dynamic analysis methodology for partially automating problem determination in Internet services, which uses cluster analysis to identify groups of components that tend to be used together in failed requests. Liu and Han [30] propose a dissimilarity metric called $R$-Proximity for comparing traces of failed executions, which regards two failing traces as similar if they suggest roughly the same fault location. They present evidence that with $R$-Proximity, failing traces due to the same fault can be grouped together to aid debugging. Mao and Lu [31] present a technique that employs Markov models of program behavior, clustering of profiles, and priority-ranked $n$-per-cluster sampling to extract a representative sample of failed executions from a set of executions. Other related work addresses the problem of correlating events generated by distributed systems. Gruschke [16] proposes an event correlation system that groups events according to information derived from a dependency graph of a distributed system. Yemini et al. [48] describe an event correlation system that employs a codebook that represents the dependency graph formed by the relations between events. Bouloutas et al. [2] describe a framework for fault localization that uses dependency graphs and heuristic algorithms for alarm correlation.
Chapter 3

A Case Study of Mislabling by Users

In the fall of 2007, we performed a modest case study involving the graduate and undergraduate software engineering classes at Case Western Reserve University that illustrated how unreliable user feedback can be. The case study examined how well users are able to identify correctly written Java source code. Each of forty-eight students was shown ten self-contained Java files via a web application and asked to determine whether or not the file was syntactically correct.

The files presented to the students were all taken from the Jacks test suite, an automated compiler test suite for the Java language [44]. The tests in Jacks fall into one of three categories:

1. Positive test cases: tests where the compiler should produce a compiled Java .class file.

2. Negative test cases: tests where the compiler should return a compiler error.

3. Tests that result in output from an executable. These tests usually compile a snippet of code, execute the resulting executable, and then verify the output.
Table 3.1: Compiler Results

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Version</th>
<th>Passed</th>
<th>Failed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Positive</td>
<td>Negative</td>
</tr>
<tr>
<td>Javac</td>
<td>1.5.0.11</td>
<td>36</td>
<td>38</td>
</tr>
<tr>
<td>Javac</td>
<td>1.6.0</td>
<td>35</td>
<td>36</td>
</tr>
<tr>
<td>GCJ</td>
<td>4.1.2</td>
<td>34</td>
<td>49</td>
</tr>
</tbody>
</table>

Only tests in the first and second categories were selected for this study. A total of one hundred test cases were used, of which forty-two were positive and fifty-eight were negative. These tests trigger real defects in commonly used Java compilers. Table 3.1 shows the numbers of positive and negative test cases that each compiler (Javac 1.5.0, Javac 1.6.0, and GCJ 4.1.2) passed and failed\(^1\). The test cases were, in part, selected so that the students could not simply compile the test cases and receive an answer. The high level of failures exhibited by each compiler is not representative of the compilers’ actual performance on the test suite.

Each test case was randomly assigned to five students. For each test case instance, the students were asked to:

1. Determine whether it would compile or not, that is, whether the file is syntactically correct. “PASS” indicates that they felt the test case was syntactically correct. “FAIL” indicates that they felt the compiler should give an error.

2. Give a reason from the Java Language Specification (JLS)[15] as to why the file does or does not compile.

To aid the students in diagnosis, they were directed to six chapters in the JLS that were relevant to the test cases; these chapters (as well as the number of positive and negative test cases selected from each) are listed in Table 3.2. The students\(^1\)

---

\(^1\)Two of the tests were not executed by any of the three compilers.
1. class T1628u1 {
2.     T1628u1 (){}
3.     void foo(int i) {
4.         switch (i) {
5.             
6.             case 0:
7.                 byte b = 0;
8.                 break;
9.             default:
10.                 b++;
11.             
12.         }
13.     }
14. }

Figure 3.1: Sample Negative Test Case from Jacks Test Suite (16.2.8-unassigned-1)

were advised that a significant portion of the test cases were not properly handled by commonly used Java compilers — either the compiler would not compile the class file, or it would fail to emit an error. The students were given two weeks to evaluate the ten test case instances assigned to them.

Figure 3.1 gives a sample question that some of the students received. This is a negative test case taken from Chapter 16, section 16.2.8 of the JLS. Both javac 5 and 6 pass this test, by giving the following compiler error: T1628u1.java:10: variable b might not have been initialized. GCJ 4.1.2 incorrectly compiled this file. The student is required to:

1. Identify the switch statement in function foo as the problematic statement.

2. The unusual part of this foo appears to be that byte b is set to 0, but this may not occur if i!=0.

3. This is related to assignment, and Chapter 16 defines Definite Assignment.

4. Section 16.2.8 and 16.2.9 of Chapter 16 define when a variable is defined in a switch statement.
Table 3.2: Chapters from JLS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Positive</th>
<th>Negative</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Conversions and Promotions</td>
<td>8</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>Packages</td>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>Classes</td>
<td>9</td>
<td>19</td>
<td>28</td>
</tr>
<tr>
<td>10</td>
<td>Arrays</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>Expressions</td>
<td>12</td>
<td>22</td>
<td>34</td>
</tr>
<tr>
<td>16</td>
<td>Definite Assignment</td>
<td>8</td>
<td>12</td>
<td>20</td>
</tr>
</tbody>
</table>

Figure 3.2: Histogram of Number of Correct Answers for each User

For purposes of analysis, any question not answered was considered to be a PASS; this occurred in forty of the evaluations. Only fifteen of these unevaluated test cases were actually correct.

The students correctly classified 324 of the 480 (67.5\%) test case instances. Figure 3.2 gives a histogram of the number of correct answers by user. The introductory programming classes at Case Western Reserve University are all taught in Java; the students therefore had at least a passing familiarity with the language. Thirty-eight of the students (79\%) correctly classified six or more test case instances.

Figure 3.3 shows the number of correct answers per test case, separated into
positive and negative test case instances. The students correctly classified 144 of the
202 positive test case instances (71.3%) as compilable. They correctly classified 180
of the 278 negative instances (64.7%) as not compilable. The students may have an
easier time identifying positive test instances; negative test instances, especially ones
that involve esoteric areas of the specification. This trend holds when examining
only the difficult questions (where “difficult” means that the test case received two
or fewer correct answers): 4% of the positive test cases and 6.5% of the negative were
difficult. At the same time, 16.8% of the positive test cases were “easy” (had more
three or more correct responses) compared with 14.3% of the negative test cases.

Figure 3.4 displays the number of correct answers per test case, broken down by
\textit{JLS} chapter. Test cases in chapters fifteen and sixteen presented the most difficulty;
students misclassified test case instances from these two chapters 42.7% and 34.4%
of the time, respectively. In comparison, test cases in the other four chapters were
misclassified between 20% and 29% of the time. The higher mislabeling found in
chapters 15 and 16 mirrors the relationship between positive and negative test cases:
64% and 50% of the difficult questions in chapters 15 and 16, respectively, were
negative test cases. In addition, 3 of the 4 difficult questions in chapter 8 were also negative test cases.

This case study illustrates an extreme case: there are approximately as many failures as successes; we expect that normal beta software will have relatively few defects. Software in beta typically performs correctly, with only a few visible defects. There should be many more reported successes than failures. The high mislabeling rate for both successes and failures can be attributed to both the tests selected from Jacks and the complexity of the JLS. Many of the tests from the Jacks test suite address unusual parts of the language specification. These tests exercise functionality that is prone to contain defects because it is difficult both for developers to implement and for users to understand. While many applications have such functionality, executions that involve only basic, commonly used functionality are presumably more numerous and much less likely to be mislabeled by users.

The study illustrates several issues with collecting feedback from users about software failures. Complex software specifications are difficult for even seasoned developers to implement (as shown by the defects chosen for this study) — for users these
same specifications will likely be inscrutable. The users in this study had advantages that real users do not: they not only had the specification for the Java language, they had specific chapters to examine that were related to the inputs. In reality, users will either not have access to the specification, or (even more likely) will not bother reading it. An excellent example of this is the web browser: modern web browsers implement multiple specifications, each of which is as complicated as the JLS. Browsers must implement multiple versions of HTML, CSS, and javascript. It is unlikely that a user will be able to distinguish between implementation defects with one of these specifications and implementation issues with a particular website. At the same time, user-focussed software will have failures and errors that are more easily identified by users.
Chapter 4

Techniques

In this chapter, we present three techniques for selecting executions for developers to review based on both user labels of SUCCESS/FAILURE and software profiles. The presented techniques address two differing views regarding software failures.

In the first scenario, developers wish to discover and fix a high number of defects. At the same time, they also find it important to discover as many as failures as possible. There are two primary reasons for doing this:

1. The defects break mission-critical features. These breakages may require developer intervention, either so developers can fix the error (possibly data loss or corruption), or so they can notify the users of a work around. This may occur even if a particular user does not report the failure.

2. Debugging a particular defect may require several execution profiles that the developer uses to identify the root cause of the defect. Developers may also be using fault localization techniques that require several examples of a defect to work. Several fault localization techniques are described in Section 2.7.

Based on one or both of these reasons, developers will want to review all executions that users have labeled FAILURE. In addition, the need to discover as many failures
as possible may prompt them to review additional executions *not* labeled FAILURE. The presented techniques seek to discover more failures with a minimal increase in executions reviewed.

In the second scenario, developers still wish to discover as many defects as possible. Here, they are not as concerned with discovering failures; they assume that the defects present are both not mission-critical and do not require advanced fault localization techniques (developers come to these conclusions given the nature of their application, and past experience with similar defects). It is also possible that the developers will lack the resources to review a large number of executions; they may decide against reviewing the larger number of executions required by the first scenario. Instead, they focus on reviewing user-labeled FAILUREs that are *most likely* to actually be failures.

### 4.1 Data Collection and Analysis

In order to apply the techniques detailed here, it is necessary to obtain four kinds of information about each execution that occurs during operational testing.

1. A user assessment (label) characterizing the execution as SUCCESS or FAILURE

2. An execution profile of suitable type

3. Audit information about inputs and outputs that is sufficient for developers to determine, if necessary, whether the execution was actually successful

4. Diagnostic information in case the execution is determined to be a failure

One way to obtain user assessments of executions is to add a feature to the software by which a user can report a failure when they believe one has occurred\(^1\). Executions

\(^{1}\)Failures caught by the software itself can be automatically labeled as such and need not be corroborated.
that are not reported to fail can be assumed to be successful. Alternatively, the user can be queried upon exiting the application about whether their run was successful and, if not, about what went wrong. With appropriate instrumentation, execution profiles can be collected online or offline. Online profiling requires instrumenting the application software (or its execution platform) for a particular form of profiling prior to deployment. On the other hand, if executions are captured online so they can be replayed offline, any form(s) of profiling can be employed during replay. Although this approach requires the deployed application or its platform to be instrumented to support execution capture, only the software or platform used during replay must be instrumented for profiling. Capture and replay of executions, where it is feasible, maximizes the amount of useful information that is available to developers for reviewing executions and for diagnosing the cause of failures.

4.2 Cluster Filtering with One-Per-Cluster Sampling

Cluster Filtering with one per cluster sampling (OPC) was originally presented in [7]. OPC clusters all executions via a clustering algorithm such as $k$-means. It randomly selects one execution from each cluster for developers to review. OPC does not take into account user labels.

4.3 Review All FAILUREs

Review All FAILUREs (RAF) is a baseline technique against which we will compare our more sophisticated techniques. RAF ignores the profile accompanying an execution, and simply prompts developers to review all executions labeled FAILURE by users. When compared with round-robin cluster filtering (described in Section 4.6),
RAF randomly orders executions for review.

4.3.1 RAF+

RAF+ is an extension to RAF that, in addition to reviewing all executions labeled FAILURE, RAF+ prompts developers to review a random selection of SUCCESS-labeled executions. In Chapter 6, corroboration-based filtering is compared with both RAF and RAF+. The number of executions slated for review by RAF+ is equivalent to a run of CBF with $T = 1$ (see Section 4.4).

4.4 Corroboration-based Filtering

Corroboration-based Filtering (CBF) is a technique for selecting executions evaluated by users for review. It is based on the conservative assumption that the potential cost of failures is very high. CBF corresponds to the first scenario, described at the beginning of this chapter; it marks executions for review if they are (1) labeled FAILURE by users and (2) there is sufficient evidence to require an execution be reviewed in spite of the fact that a user has labeled it SUCCESS. It works by attempting to corroborate an execution’s label (whether that label is SUCCESS or FAILURE). If an execution $e$ is labeled SUCCESS by the user, then developers should be able to corroborate that label by looking at the labels of similar executions (where similarity is determined by a clustering algorithm). If other users have all labeled their executions SUCCESS as well, then developers have corroboration for the successful label and do not have to worry about reviewing it. On the other hand, $e$ is suspect if a user has labeled it FAILURE, there is a similar execution labeled FAILURE, or if its label cannot be corroborated ($e$ is in a small or singleton cluster).
4.4.1 Filtering Rules

Suppose that the test executions have been clustered based on their profiles. Let $C$ be any cluster. If $|C|$ (the size of $C$) does not exceed a small corroboration threshold $T$, all of the executions in $C$ are reviewed by developers, even if they are all labeled SUCCESS. Otherwise, executions that users have labeled FAILURE are reviewed by developers to determine if they are actual failures. If any execution labeled FAILURE in cluster $C$ is confirmed to be an actual failure then all of the executions in $C$ are reviewed by developers, regardless of label. However, if a cluster $C$ only contains executions labeled SUCCESS or if all the executions in $C$ labeled FAILURE are determined to be successful by developers, then the behavior represented by $C$ is considered validated, and does not require additional developer review.

Corroboration-based filtering can be thought of as implicitly classifying clusters as SMALL, SUSPECT, OK, or BAD. SMALL and SUSPECT are temporary classifications; OK and BAD are final ones. A cluster $C$ is classified as SMALL if $|C| \leq T$. All of the executions in a SMALL cluster must be reviewed by developers, because even if it contains only SUCCESS labels it is considered too small to adequately corroborate them. $C$ is classified as SUSPECT if it contains one or more FAILURE labels. All of the executions in a SUSPECT cluster that are labeled FAILURE must be reviewed by developers to determine if they are actual failures. If developers find that any execution in a SMALL or SUSPECT cluster $C$ is an actual failure then $C$ is reclassified as BAD; if no actual failures are found then $C$ is reclassified as OK. $C$ is also classified as OK if $|C| > T$ and $C$ contains only SUCCESS labels. When a cluster is classified as OK, the behavior it represents is considered validated and no un-reviewed executions in it need be reviewed by developers. When a cluster $C$ is classified as BAD, all unreviewed executions in it must be reviewed, because they have profiles similar to one or more actual failures discovered in $C$ and hence may fail for similar reasons.
Corroboration-based filtering is inspired by the fact that studies of cluster filtering have shown that when tests are clustered based on execution profiles alone — without labels indicating whether the tests were successful or unsuccessful — failing tests tend to be placed in small clusters or singletons [7, 8]. Note, however, that with CBF, SUCCESS/FAILURE labels assigned to similar tests by different users provide independent checks on whether individual clusters represent correct program behaviors or not, and they serve to corroborate or contravene individual user assessments of particular executions.

There are three reasons for reviewing all of the executions in a cluster if it is confirmed to contain an actual failure. First, our previous empirical studies of failure pursuit sampling[8], an extension of cluster filtering that explores the nearest neighbors of a confirmed failure to find other failures, indicated that neighboring failures are sometimes caused by different defects [27, 28]. This may happen, for example, because the defects occur in the same region of code or involve the same program feature. Second, examination of additional instances of failures caused by a given defect may facilitate debugging by providing more clues about their common cause. Third, the relative frequency of operational failures caused by particular defects is an indication of the individual defects’ impact on users and is therefore helpful to consider in prioritizing defects for repair.

### 4.4.2 Probabilistic Analysis

We employ three principal metrics in evaluating corroboration-based filtering: the number $F_d$ of actual failures discovered by developers; the number $D_d$ of distinct defects consequently discovered (assuming successful debugging); and the number $R$ of executions reviewed by developers. Let $F_a$ be the total number of actual failures among the entire set of test executions, and let $D_t$ be the total number of underlying defects responsible for these failures. ($F_a$ and $D_t$ are known in the empirical study
reported in Chapter 5, but they are generally unknown in intended applications of CBF.) Then the ratios $F_d/F_a$ and $D_d/D_t$ are measures of the effectiveness of CBF for revealing failures that occurred during operational testing and the defects that caused them, respectively. $R$ is a measure of the manual effort entailed by CBF. The expected values of $F_d$ and $R$ depend on the probabilities with which different executions are mislabeled by users. With particular software under test, it is possible that actual failures are more likely to be mislabeled than actual successes or vice versa.

For any cluster $C$, let $f(C)$ denote the number of actual failures in $C$, $0 \leq f(C) \leq |C|$, and let $p_{ml|f(C)}$ denote the mislabeling probability for actual failures in $C$, and let $p_{ml|s(C)}$ be the mislabeling probability for actual successes in $C$. If $|C| \leq T$ then each execution $e \in C$ is reviewed with probability 1. Suppose that $|C| > T$, and let $e \in C$. We distinguish two cases:

1. $e$ is an actual success. The probability $p_{r|s(C)}$ that $e$ is reviewed in this case is the probability that either (a) $e$ is mislabeled or (b) $e$ is correctly labeled and at least one of the actual failures in $C$ is correctly labeled. Since each execution is labeled independently of others, we have:

$$p_{r|s(C)} = p_{ml|s(C)} + (1 - p_{ml|s(C)})(1 - p_{ml|f(C)})$$  \hspace{1cm} (4.1)

Thus if $C$ contains no actual failures then $p_{r|s(C)} = p_{ml|s(C)}$ and $p_{r|s(C)}$ increases with the number of actual failures in $C$ (provided $0 \leq p_{ml|f(C)} < 1$).

2. $e$ is an actual failure. The probability $p_{r|f(C)}$ that $e$ is reviewed in this case is the probability that:

$$p_{r|f(C)} = 1 - p_{ml|f(C)}$$  \hspace{1cm} (4.2)

Thus, the more actual failures there are in a cluster, the more likely they are to
be reviewed by developers (provided $0 < p_{m||f(C)} < 1$).

These equations indicate that if executions can be clustered so that actual successes are grouped together and are mostly separated from actual failures, CBF will lead developers to review most actual failures but relatively few actual successes.

For any cluster $C$ with $|C| > T$, the number of actual failures reviewed by developers is the sum of $f(C)$ Bernoulli random variables $X_i$ with $P(X_i = 1) = p_{r|f(C)}$. Similarly, the number of actual successes reviewed is the sum of $|C| - f(C)$ Bernoulli random variables $Y_j$ with $P(Y_j = 1) = p_{r|s(C)}$. Therefore, the expected number of executions reviewed by developers is given by the equation:

$$E(R) = \sum_{C: |C| \leq T} |C| + \sum_{C: |C| > T} (f(C)p_{r|f(C)} + (|C| - f(C))p_{r|s(C)})$$ (4.3)

Assuming that developers discover failures when they review them, the expected number of discoveries is:

$$E(F_d) = \sum_{C: |C| \leq T} f(C) + \sum_{C: |C| > T} f(C)p_{r|f(C)}$$ (4.4)

Since the number $D_d$ of defects discovered depends on a possibly complex relationship between defects and failures, we do not attempt to provide a formula for its expected value, although $D_d$ values from our empirical study are characterized in Section 6.2.

### 4.5 Review-All-failures and $k$-Nearest Neighbors

Review-All-FAILURES and $k$-Nearest Neighbors (RAF+$k$NN) is designed to be an alternative to Corroboration-Based Filtering, that is less expensive but reveals nearly $\text{2}$ 

Note that in general these random variables are not independent, because all executions in $C$ are reviewed if any actual failure is reviewed.
as many failures and defects. As noted in Section 4.4, CBF slates a great many executions for review under the assumption that the potential cost of failures is very high. CBF therefore calls for developers to review executions with user label SUCCESS if they have unusual profiles or they are in a cluster with a confirmed failure.

RAF$+k_{NN}$ is an extension of review-all-FAILUREs (RAF). The experiments detailed in Chapter 6 demonstrate that while RAF is less expensive than CBF (in terms of executions reviewed), it also reveals considerably fewer failures. RAF$+k_{NN}$ starts by computing the $k$ nearest neighbors of each execution. Then, like CBF and RAF, it assumes that all executions labeled FAILURE by users are suspicious. After developers review these FAILURE-labeled executions (and confirmed some of them to be actual failures), RAF$+k_{NN}$ prompts them to review the $k$-nearest neighbors of each confirmed failure.

RAF$+k_{NN}$ is similar to CBF in that both classify a subset of the SUCCESS-labeled executions as “suspicious”. For CBF, the suspicious executions are:

1. Executions labeled FAILURE by users.
2. Executions in the same cluster as a confirmed failure.
3. Executions with unusual profiles (defined as executions in clusters where $|C| \leq T$ — the corroboration threshold).

RAF$+k_{NN}$ reviews the first class, a limited portion of the second class, and it completely ignores the third class. Because the set of $k$ nearest neighbors does not take into account distance, it is possible that RAF$+k_{NN}$ will group executions together that would not be grouped together by a clustering algorithm (such as $k$-means) — in particular executions in small clusters will be grouped together. In practice, this rarely occurred for even high ($k = 5$) values of $k$. By ignoring unusual executions and only reviewing the $k$-nearest executions to a confirmed failure, we hope that
RAF+$k$NN will find similar numbers of failures and defects to CBF while requiring fewer executions reviewed.

### 4.6 Round-robin Cluster Filtering with $k$-NN Ranking

Round-robin Cluster Filtering with $k$-NN Ranking (RRCS) has a different focus from both CBF and RAF+$k$NN. Whereas CBF and RAF+$k$NN are both meant for our first scenario (where developers are willing to review extra executions because the cost of both failures and defects is very high), RRCS targets the second scenario: developers wish to discover as many defects as quickly as possible. Here, developers either expect that the encountered defects are unlikely to cause catastrophic failures or do not have the personnel or time necessary to review large quantities of executions. Instead, like RAF, RRCS only reviews executions labeled FAILURE by users. However, where RAF randomly selects FAILUREs for review, RRCS organizes them into clusters and then (optionally) ranks the executions in each cluster by proximity to confirmed failures.

RRCS takes all executions labeled FAILURE and clusters them using a clustering algorithm such as $k$-means (this differs from CBF, which clusters all executions regardless of label). For each cluster, $C$, randomly assign a rank ($R$), from $\{1, ..., |C|\}$, to each $e \in C$. Developers then review all executions with a particular rank, starting with 1 and ending with $\max(\{|C_1|, ..., |C_k|\})$, until all executions are exhausted. This more systematic exploration of the profile space should reveal defects more quickly when compared with RAF’s random ordering, assuming that:

1. failures with the same underlying defect are grouped into relatively few clusters
2. there are few small clusters
4.6.1 \( k \)-NN Ranking

In addition to grouping FAILURE-labeled executions into clusters, we consider an extension to RRCS which, for a cluster \( C \), ranks each execution \( e \in C \) according to their proximity to confirmed failures (where a confirmed failure \( f_c \in C \)).

To use \( k \)-NN ranking, developers use RRCS to assign a rank to each execution (as described above), and then review all executions with a rank less than or equal to some threshold \( N \). Once the first \( N \) executions in each cluster have been reviewed, the remaining, un-reviewed, executions are re-ranked using a \( k \)-nearest neighbor algorithm. For each unconfirmed execution \( E_u \in C \), we find the \( k \) nearest reviewed executions in \( C \). Each execution \( E_u \) is ranked via the number of confirmed failures within the \( k \) reviewed executions. The un-reviewed executions are reviewed using this new ranking. As new failures are confirmed the ranking for the remaining \( E_u \in C \) is updated. \( k \)-NN ranking is not used with any cluster with \( |C| < N \).

4.7 Assumptions

Several assumptions underlie these techniques: most test executions are actually successful; users don’t make too many errors in assessing and labeling executions; actual successes are clustered together and are mostly separated from actual failures; and there aren’t too many small clusters. If these conditions are met, these techniques should lead developers to review more actual failures and fewer actual successes, producing more accurate assessments of operational reliability\footnote{These may or may not involve statistical estimates of reliability. How such estimates should be made is not addressed in this dissertation.} than conventional beta testing, at less cost than with cluster filtering.
Chapter 5

Experimental Methodology

In order to evaluate the potential utility of the presented techniques, we conducted three empirical studies in which each technique was applied to all six data sets. The studies evaluated the techniques using three primary metrics:

- How effective is the technique, as measured by the number $F_d$ of actual failures discovered?
- How many actual defects were discovered ($D_d$) by the technique?
- How costly is the technique, as measured by the number $R$ of executions reviewed by developers?

For each of the test sets, we knew which executions actually failed. However, we did not have users’ assessments of those executions to employ in labeling them as SUCCESSes and FAILUREs; moreover, we wished to vary the probability of mislabeling, in order to study its effects on the number of defects discovered, failures discovered, and executions reviewed.\(^1\) Therefore, we simulated user labeling by randomly mislabeling some executions in each test set. Each experiment consists of a

\(^1\)Note that different applications are likely to have different mislabeling probabilities.
Table 5.1: Subject Data

<table>
<thead>
<tr>
<th>Program</th>
<th>Version</th>
<th>LOC</th>
<th>Runs</th>
<th>Failures</th>
<th>Defects</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>2.95.2</td>
<td>330,000</td>
<td>3,333</td>
<td>136</td>
<td>26</td>
</tr>
<tr>
<td>Javac</td>
<td>1.31-02-b02</td>
<td>20,000</td>
<td>3,140</td>
<td>233</td>
<td>67</td>
</tr>
<tr>
<td>JTidy</td>
<td>3</td>
<td>13,851</td>
<td>7,990</td>
<td>308</td>
<td>8</td>
</tr>
<tr>
<td>ROME</td>
<td>11/2005</td>
<td>10,273</td>
<td>8,000</td>
<td>2,390</td>
<td>8</td>
</tr>
<tr>
<td>Xerces</td>
<td>2.0.0</td>
<td>64,735</td>
<td>57,780</td>
<td>20,954</td>
<td>17</td>
</tr>
</tbody>
</table>

technique, associated variables, and a mislabeling. Each experiment was repeated 100 times, and the mean metric values were computed.

5.1 Subject Programs

In this study, we used five subject programs, detailed in Table 5.1. Two of these data sets, GCC and Javac, use test suites created by the project developers; the other three test data sets, JTidy, ROME, and Xerces, were created using randomly collected files that should better mirror operational inputs.

Version 2.95.2 of the GCC C compiler [11] was executed on a subset of the regression test suite for GCC 3.0.2, which included tests for defects still present in version 2.95.2. Only tests which executed compiled code were used. These tests were all self-validating. GCC was profiled using the GNU test coverage profiler, Gcov.

Javac 1.31-02-b02 [14] was executed on the Jacks compiler test suite [44]. Jacks is a compiler test suite that evaluates compliance against the Java Language Specification. The failures in GCC and Javac were manually classified as described in [38].

Version 3 of JTidy [24] was instrumented with five failure checkers as described in [10]. JTidy was executed on 7,990 HTML and XML files randomly gathered from Google Directory [12] with a web crawler.
Both *Javac* and *JTidy* were profiled using custom profilers that recorded function-call execution counts.

The *ROME* RSS/Atom parser [42] was downloaded in November 2005 from the main development branch of the public *CVS* repository. Failure checkers for eight defects were inserted. This instrumented version was executed on 8,000 Atom and RSS files downloaded from *Google Search* results [13] using a custom web crawler. Of the 8,000 executions, 2,390 of them were failures. *ROME* was instrumented for profiling using two different profilers: the *Java Interactive Profiler* [21] was used to collect function-call counts, and *Cobertura* [9] was used to collect line coverage counts. In the experimental results, *ROME* is labeled *ROME-F* (for function-call profiles) or *ROME-L* (for line-coverage profiles). *ROME* provides a small case study that illustrates the effects of choosing one profiler over another.

*Xerces 2* [43] was downloaded from the version 2.0.0 tag of the public *Subversion* repository. Eight failure checkers were inserted. The defects in *Xerces* involved the DOM and SAX APIs, XML validation, and XML serialization. A total of 9,630 files were collected from the system directories of an Ubuntu Linux 7.04 [45] machine and from *Google Search* results [13]. *Xerces* was executed on each file with six different combinations of flags selecting particular features. This resulted in 57,780 executions and 20,954 failures.

Both *ROME* and *Xerces* had substantially more failures than the other data sets. In addition, *Xerces* had 57,780 executions, making even individual experiments taxing for modern computers. We expect that most beta testing will contain mostly successful executions and relatively few actual failures. In line with these expectations, and to keep the run time of individual experiments low, we selected a random subset of 5,000 executions for both *ROME* and *Xerces*, which included 250 failures each. Table 5.2 lists the number of executions, failures, and defects used in all experiments.

A number of the failing executions for the operational data sets triggered multiple
Table 5.2: Experimental Data

<table>
<thead>
<tr>
<th>Program</th>
<th>Executions</th>
<th>Failures (Percent)</th>
<th>Defects</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>3,333</td>
<td>136 (4.1%)</td>
<td>26</td>
</tr>
<tr>
<td>Javac</td>
<td>3,140</td>
<td>233 (7.4%)</td>
<td>67</td>
</tr>
<tr>
<td>JTidy</td>
<td>7,990</td>
<td>308 (3.9%)</td>
<td>8</td>
</tr>
<tr>
<td>ROME-F 5%</td>
<td>5,000</td>
<td>250 (5.0%)</td>
<td>7</td>
</tr>
<tr>
<td>ROME-L 5%</td>
<td>5,000</td>
<td>250 (5.0%)</td>
<td>7</td>
</tr>
<tr>
<td>Xerces 5%</td>
<td>5,000</td>
<td>250 (5.0%)</td>
<td>17</td>
</tr>
</tbody>
</table>

defects. The reasoning behind this decision is that defects triggered in combination likely represents a specific behavior by users. This is born out by the fact that certain defect combinations occur more often than others and that some defects only occurred in combination with others. In the experiments, each unique combination of defects was treated as a distinct defect; this resulted in eight, four, and fourteen such combinations for JTidy, ROME, and Xerces, respectively.

5.2 Data Analysis

All data analysis was implemented using the GNU R statistical package [39]. The \textit{k-means} clustering algorithm was used to cluster executions for CBF and RRCS. To compute the \textit{k}-nearest neighbors, the \textit{knnFinder} package [26] was used for RAF+$k$NN and the \textit{yaImpute} [5] package was used for RRCS. For a dissimilarity metric, we used the \textit{proportional-binary} metric [27], which is based on Euclidean distance and takes into account both whether a program element was covered at least once and how many times the element was executed. If we define $C_{i,j}$ as the count of the number of times that element $j$ was executed on test run $i$, define $P_{i,j}$ as:

$$P_{i,j} = \frac{C_{i,j} - \min_k C_{k,j}}{\max_k C_{k,j} - \min_k C_{k,j}} \quad (5.1)$$

Thus, the $P_{i,j}$ are normalized counts. Also, define $B_{i,j}$ to be 0 if $C_{i,j} = 0$ and to
be 1 otherwise. Then the proportional-binary dissimilarity between runs \( n \) and \( m \) is:

\[
D_{n,m} = \sqrt{\sum_k (P_{n,k} - P_{m,k})^2 + |B_{n,k} - B_{m,k}|}
\]  

(5.2)

The plots produced in this dissertation were created using the \textit{ggplot2} [47] and \textit{reshape} [46] packages.

### 5.3 Study of Corroboration-based Filtering

This empirical study attempts to establish that CBF helps developers to discover more failures and defects than simpler techniques while requiring only a modest increase in reviewed executions. CBF augments the FAILURE-labeled executions provided by users with SUCCESS-labeled executions that are suspicious, either because they are in the same cluster as a confirmed failure, or because they are in a very small cluster (defined by the value of \( T \)).

For each subject program and accompanying test set, the mean number of failures discovered \( (F_d) \), defects discovered \( (D_d) \), and executions reviewed \( (R) \) were obtained for three different clusterings of the test executions (sized at approximately 10%, 20%, and 30% of the original test set) and for \( T = 1, 2, \ldots, 5 \). For comparison, the same measures were also obtained for three alternative techniques:

- Cluster filtering with one per cluster sampling (OPC — described in Section 4.2). OPC calls for selecting one execution at random from each cluster and does not take into account user labeling.

- Review-All-FAILUREs, which reviews all executions labeled FAILURE. RAF is described in Section 4.3.

- RAF+: an extension to RAF which supplements the executions reviewed by RAF with SUCCESS-labeled executions so that \( R \) for RAF+ matches \( R \) for
CBF with $T = 1$. Thus, for each clustering used by CBF, there is a corresponding RAF+ (RAF+ 10%, RAF+ 20%, and RAF+ 30% correspond to CBF10+1, CBF20+1, and CBF30+1, respectively). RAF+ should establish whether or not CBF performs better than RAF (in terms of $F_d$ and $D_d$) simply because it reviews more executions. RAF+ is described in Section 4.3.1.

In each iteration of the experiments in this study, each execution was mislabeled with a fixed probability $p_{ml}$, regardless of whether it was an actual success or failure. This probability was varied between 0.0 and 0.3 in repeated experiments.

The results from the empirical evaluation of CBF are presented in Chapter 6.

### 5.4 Study of RAF+$k$NN

The empirical evaluation of CBF (results in Chapter 6) demonstrated that while CBF leads developers to discover a high percentage of failures and defects, it also entails reviewing a large number of executions. RAF+$k$NN was developed, in part, to allow developers to review fewer executions than CBF, but still discover a high percentage of failures and defects.

To study RAF+$k$NN, we computed $F_d$, $D_d$, and $R$ for $k = 1, 2, \ldots, 5$. RAF+$k$NN was compared to CBF with $T = 1$ and RAF. The previous study (Section 5.3) explored the effect of users uniformly mislabeling executions, regardless of status. This scheme, dubbed “uniform mislabeling”, assumes that users are equally likely to mislabel failures and successes. With uniform mislabeling, executions are mislabeled with a single probability $p_{ml}$ that is varied systematically between 0.0 and 0.3. It is suspected, however, that for many applications, users are much more likely to mislabel actual failures than successes. Most successful executions represent common use cases that users are less likely to mislabel. Conversely, failed executions tend to be found in less common functionality that users are more likely to mislabel. To study
this possibility, we also employed **differential mislabeling**, in which actual successes are randomly mislabeled with a fixed probability $p_{ms} = 0.10$ and actual failures are randomly mislabeled with a probability $p_{mf}$ that is varied between 0 and 0.3.

The results for the evaluation of RAF+$k$NN are presented in Chapter 7.

### 5.5 Study of Relative Costs of CBF, RAF, and RAF+$k$NN

In addition to comparing $R$, $F_d$ and $D_d$ for RAF+$k$NN, RAF, and CBF, we also conducted an analysis of the **relative costs** of these three techniques. This experiment explores the trade-off between the amount of time developers spend reviewing executions and the cost of leaving defects undiscovered. By relating developer time with the cost of undiscovered defects, developers can select an appropriate technique based on prior defect cost. This study is presented in Chapter 8.

### 5.6 Study of RRCS with $k$-NN Ranking

The study of relative costs in Chapter 8 demonstrates that it can be advantageous for an organization to use RAF over CBF and RAF+$k$NN when defects are inexpensive relative to developer time. Under these conditions, it is not cost effective for the organization to discover all defects. Instead, developers should focus only on those defects that have been reported by users (this is equivalent to the set of defects with at least one execution labeled FAILURE by users).

CBF and RAF+$k$NN, in essence, provide a way for developers to discover mislabeled failures. RRCS provides a way for developers to minimize the number of mislabeled successes they review.

In this study, we compare RRCS and RAF. RRCS reviews the same set of ex-
ecutions as RAF, but in a systematic manner. To compare the two techniques, we
examine the following metrics:

- What is the mean number of executions developers must review to discover $F$
  failures?
- What is the mean number of executions developers must review to discover $D$
  defects?

For each data set, we computed these two measures for RRCS using $k = \{0, 1, 3\}$
(recall that $k = 0$ denotes RRCS without $k$-NN ranking, or “plain RRCS”). The
$k$-means clustering algorithm was used to cluster FAILUREs. The number of clusters
was set to 10% of the number of FAILURE-labeled executions. For each experiment,
we computed the minimum number of reviews required to find $N$ failures, where
$N = 1, 2, \ldots, max(F)$ or $N$ defects where $N = 1, 2, \ldots, max(D)$.

Both uniform and differential mislabeling were used to study RRCS and RAF.
For uniform mislabeling, $p_{ml}$ was varied from 0.05 to 0.25 in increments of 0.05. For
differential mislabeling, $p_{mf}$ was varied from 0.05 to 0.25 in increments of 0.05 and
$p_{ms} = p_{mf}/2$.

The experimental results for this study are shown in Chapter 9.
Chapter 6

Evaluation of Corroboration-Based Filtering

This chapter presents the results of the empirical study examining corroboration-based filtering (CBF), which is described in Section 5.3. This study compares CBF with the simple review-all-FAILUREs strategy (and its extension, RAF+), and cluster filtering with one-per-cluster sampling (OPC). These techniques are described in Sections 4.4, 4.3, and 4.2, respectively.

Section 6.1 analyzes the number of failures discovered by applying the given techniques for differing values of $p_{ml}$. Section 6.2 analyzes the number of unique defects discovered. Section 6.3 gives the number of executions reviewed, and how many executions developers will review to discover a single failure. Finally, Section 6.4 analyzes CBF with no threshold value; CBF with $T = 1$ is compared to CBF with $T = 0$.

6.1 Failures Discovered

CBF discovered more failures than the naive RAF technique for all six datasets. In addition, increasing the number of clusters and using higher values of the threshold ($T$) allowed developers to discover more failures. When compared with the RAF+
technique (which reviews a random sampling of executions labeled SUCCESS in addition to all FAILURE-labeled executions), CBF with $T = 1$ discovers significantly more failures even though the two techniques review the same number of executions. This indicates that CBF does not outperform RAF by simply reviewing more executions.

For the different subject programs and clusterings, the mean number of actual failures discovered over 100 replications of corroboration-based filtering with a given mislabeling probability $p_{ml}$ tended to be greater for higher values of the threshold $T$. The number of failures discovered tended to gradually decrease in a linear fashion as $p_{ml}$ increased from 0 to 0.3.

For all data sets, increasing the number of clusters caused $F_d$ to vary with $T$. Figure 6.1 illustrates this trend. When the number of clusters is 10% of the total executions, the difference between mean failures discovered for $T = 1$ and $T = 5$ is 0.13 failures (at $p_{ml} = 0.2$). For 20% and 30% clusters this difference increases to 4.33 and 8.14, respectively. The minimum number of failures found by CBF for GCC is 92.2% (125.42 failures).

- For GCC, the difference in failures discovered between $T = 1$ and $T = 5$ was 1.38 for the 10% clustering, and increased to 13.47 for the 30% clustering (at $p_{ml} = 0.3$).

- For Javac, the difference between $T = 1$ and $T = 5$ increased from 4.18 to 14.26 (from the 10% clustering to the 30% clustering).

- For JTidy, the difference between $T = 1$ and $T = 5$ was 5.54 with the 10% clustering, and increased to 12.98 at the 30% clustering.

- For ROME-F 5%, the difference between 10% and 30% clustering increased by 20.77 (from a difference of 5.67 at 10% clustering and 26.44 for the 30%
Figure 6.1: % Mean Failures Discovered vs. $p_{ml}$ for differing $T$ values
clustering). By comparison, *ROME-L 5%* changed only 4.28 (from 3.77 to 8.05).

- The clustering varied the least for *Xerces 5%*: the difference between $T = 1$ and $T = 5$ was only 0.84 for the 10% clustering and 4.2 for the 30% clustering.

Figure 6.2 shows the percentage failures discovered for differing values of $T$. These percentages represent the number of failures that are *always* discovered for a particular $T$ value. Mislabeling does not matter because the executions in clusters with $|C| \leq T$ are reviewed regardless of label. This figure corroborates the results shown by Figure 6.1: $T = 5$ has little effect on any of the datasets with the 10% clustering because, at best 30% of the failures are in clusters that are at or below the corroboration threshold. As the numbers of clusters increases, the number of failures developers are guaranteed to discover for any $T$ value tends to increase.

It is important to note, however, that simply raising $T$ is not the most efficient way to discover failures, even though higher values of $T$ can guarantee a certain level of failures discovered: depending on the value of $p_{ml}$, many of these “guaranteed” failures may be discovered with lower threshold values. Given a cluster, $C$, with $N$ failures, the probability that $C$ will not be marked for review by CBF is $p_{ml}^N$. For example, with $p_{ml} = 0.3$ and a cluster with two failures, the probability that this cluster will not be reviewed is only $0.3^2 = 0.09$; this probability decreases the more failures there are in a cluster.

CBF is dependent on the quality of the underlying clustering algorithm. *ROME-F* and *ROME-L* provide an interesting comparison as the two datasets are based on the same input set, but use different profiling techniques to record data (*ROME-F* uses function-call profiling and *ROME-L* uses line-coverage profiling). Because of this the relative quality of the clustering is very different. With $T = 1$, CBF with function-call profiling leads developers to discover no better than 90% of the failures; with the 30% clustering, it discovers only 85% of the failures. On the other hand, with
Figure 6.2: % Failures Discovered for Different T Values
line-coverage profiling, CBF leads developers to discover 95% or more failures. The difference here is not the number of singleton clusters (since $T = 1$): $ROME-F$ has 0, 5, and 15 failing singleton clusters and $ROME-L$ has 3, 11, and 27 for the 10%, 20%, and 30% clusterings. Instead, this can be traced back to the quality of the clustering provided by $k$-means: with $k$ equal to 10% of the total executions ($k = 500$), $ROME-F$ placed the 250 failures in 120 clusters and $ROME-L$ placed them in 59 clusters. With $ROME-F$, 244 of the failures (97.6%) were placed in 118 mixed clusters with a mean size of 13.5. This suggests that many of the clusters have only 2 failures. With $ROME-L$, on the other hand, 163 of the failures (65%) are in 20 homogenous clusters. The mean size of these clusters was 7.22. It is far more likely that a cluster consisting of 7 failures will be reviewed by developers than a cluster with only two failures. This quality difference between $ROME-F$ and $ROME-L$ remains for larger $k$ values:

- With $k = 1000$, actual failures were spread amongst 153 clusters with $ROME-F$. Only 9 clusters were homogenous, and they contained 21 (8.4%) failures. Using $ROME-L$, the failures were spread among 76 clusters, and 207 (82.8%) of them were in 50 homogenous clusters.

- With $k = 1500$, the actual failures were spread among 170 clusters for $ROME-F$ and 93 clusters for $ROME-L$. Only 14% of the failures were in homogenous clusters for $ROME-F$, compared to 92.8% of the failures for $ROME-L$.

The differences between $ROME-F$ and $ROME-L$ will be examined further as we look at the mean executions reviewed and defects discovered.

With Xerces, CBF tended to find all failures regardless of mislabeling rate and clustering. CBF discovered at least 98.3% (245.8/250) of all failures (at 30% clustering, $T = 1$, $p_{ml} = 0.3$). As with $ROME-L$ and $ROME-F$, this is related to how well the data is clustered: 87.2%, 98.4%, and 96.8% of the failures were found in ho-
Table 6.1: Mean failures discovered with OPC

<table>
<thead>
<tr>
<th>Program</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>13.98</td>
<td>26.04</td>
<td>48.16</td>
</tr>
<tr>
<td>Javac</td>
<td>30.36</td>
<td>58.51</td>
<td>88.85</td>
</tr>
<tr>
<td>JTidy</td>
<td>23.58</td>
<td>43.80</td>
<td>63.41</td>
</tr>
<tr>
<td>ROME-F 5%</td>
<td>23.88</td>
<td>49.72</td>
<td>71.75</td>
</tr>
<tr>
<td>ROME-L 5%</td>
<td>34.83</td>
<td>60.07</td>
<td>84.84</td>
</tr>
<tr>
<td>Xerces 5%</td>
<td>25.42</td>
<td>53.81</td>
<td>67.36</td>
</tr>
</tbody>
</table>

hogenous clusters (mean cluster size 9.48, 4.64, and 3.67) with \( k = \{500, 1000, 1500\} \), respectively.

Table 6.1 shows the mean failures discovered values obtained via one-per-cluster sampling for each clustering on each of the six datasets. These values are much less than the corresponding ones for CBF, even for \( p_{ml} = 0.3 \). (Note that \( p_{ml} \) is not relevant to OPC, which does not consider user assessments.)

Figure 6.3 shows CBF with \( T = 1 \) compared with RAF and RAF+ for all three clusterings. RAF+ reviews all labeled FAILUREs, and then randomly reviews labeled SUCCESSes until it has reviewed as many total executions as the equivalent CBF (ex: RAF+ 10% reviews as many executions as CBF with \( T = 1 \) and 10% clustering for the same user labeling). It is clear that, though RAF+ provides a modest improvement over RAF, it does not discover nearly as many failures as CBF. This is in spite of the fact that RAF+ and CBF review the same number of executions.

RAF+ found, at best, 6.99% more failures than RAF \( (p_{ml} = 0.3, 10\% \text{ clustering}, ROME-F 5\%) \). With Javac, JTidy, ROME-L, and Xerces, CBF found at least 19% more failures than RAF+. With GCC and ROME-F, CBF led developers to discover only 8.99% and 10.80% more failures than RAF+, respectively.
Figure 6.3: % Mean Failures Discovered vs. $p_{ml}$ for CBF, RAF+, RAF
6.2 Defects Discovered

CBF with $T = 1$ allowed developers to discover almost all of the defects present in each subject application. CBF found more defects than RAF, RAF+, and the one-per-cluster sampling method (which ignores mislabeling).

Table 6.2: Percentage mean defects discovered with $p_{ml} = 0.3$

<table>
<thead>
<tr>
<th>Program</th>
<th>CBF</th>
<th>RAF</th>
<th>RAF+</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10%</td>
<td>20%</td>
<td>30%</td>
</tr>
<tr>
<td>GCC</td>
<td>98.23</td>
<td>98.54</td>
<td>98.00</td>
</tr>
<tr>
<td>Javac</td>
<td>88.24</td>
<td>90.01</td>
<td>89.81</td>
</tr>
<tr>
<td>JTidy</td>
<td>98.88</td>
<td>99.38</td>
<td>98.38</td>
</tr>
<tr>
<td>ROME-F 5%</td>
<td>96.43</td>
<td>96.29</td>
<td>92.57</td>
</tr>
<tr>
<td>ROME-L 5%</td>
<td>92.00</td>
<td>95.29</td>
<td>95.86</td>
</tr>
<tr>
<td>Xerces 5%</td>
<td>98.82</td>
<td>99.41</td>
<td>99.41</td>
</tr>
</tbody>
</table>

With GCC, CBF would lead developers to discover 98.54% of all defects for the 20% clustering; an improvement over RAF by 1.31%. With Javac (which had 67 defects), CBF improved on both RAF and RAF+ by 6.77% and 3.47%, respectively. CBF found 2.88% and 2.38% more defects than RAF and RAF+ for JTidy. For ROME, CBF found 4.86% more defects than RAF with functional-call profiling and 4.72% with line-coverage profiling. When compared with RAF+, CBF found 2.86% and 4.72% more defects, respectively. With Xerces, CBF found 6.65% more defects than RAF and 6.06% more defects than RAF+. Because even high mislabeling ($p_{ml} = 0.3$) still means that 70% of failures will be reviewed using RAF, we can expect that most common defects will be discovered. Rarer defects are more likely to have all their occurrences mislabeled, and so are not found with RAF.

One-per-cluster sampling, as seen in Table 6.3, discovers significantly fewer defects than CBF. At best, OPC discovered 92.57% of defects (for ROME-L with 30% clustering); 3.29% fewer defects than CBF. For JTidy, CBF discovers 8.25% more de-
Table 6.3: Percent mean defects discovered with OPC

<table>
<thead>
<tr>
<th>Program</th>
<th>Clustering</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10%</td>
</tr>
<tr>
<td>GCC</td>
<td>45.08%</td>
</tr>
<tr>
<td>Javac</td>
<td>45.78%</td>
</tr>
<tr>
<td>JTidy</td>
<td>82.50%</td>
</tr>
<tr>
<td>ROME-F 5%</td>
<td>68.57%</td>
</tr>
<tr>
<td>ROME-L 5%</td>
<td>83.00%</td>
</tr>
<tr>
<td>Xerces 5%</td>
<td>73.29%</td>
</tr>
</tbody>
</table>

fects than OPC. CBF does even better with GCC, Javac, ROME-F, and Xerces: it discovers 14.42%, 19.86%, 18.57%, and 16.65% more defects than OPC, respectively.

6.3 Executions Reviewed

Though CBF finds more failures and defects than RAF, it does so at increased cost. CBF requires developers to review far more executions than RAF. As the threshold \((T)\) increases, developers may be forced to review more than 80% of all of the executions. However, because CBF allows developers to discover more failures than RAF, CBF requires fewer reviews per failure discovered than RAF, in particular with \(T = 1\).

For the different subject programs and clusterings, the mean number of executions reviewed for CBF increased linearly with both \(p_{ml}\) and \(T\). The slope of the increase with \(p_{ml}\) changes little for differing values of \(T\). This is illustrated by Figure 6.4, which shows the three clusterings for all datasets with \(T = 1, \ldots, 5\).

Increasing the value of \(T\) causes more executions to be reviewed by requiring that larger clusters be reviewed regardless of label. When the number of clusters is small (for example, the 10% clustering), there are fewer small clusters. The 10% clustering for all 6 datasets demonstrates this: there is very little difference for \(T = 1, 2, 3\). For
Figure 6.4: % Mean Executions Reviewed vs. $p_{ml}$ for CBF for differing $T$ values
the 20% and 30% clusterings, there are more singleton and small clusters for review. This also affects the number of failures discovered: as seen in Figure 6.1, lower values of $T$ (1…3) perform very similarly to each other at 10% clustering. Only the higher values (4, 5) differentiate themselves.

The difference between $T = 1$ and $T = 3$ is no more than 3% for the 10% clustering for all of the datasets with $p_{nl} = 0.3$. This increased to 10% and 20% for the 20% and 30% clusterings, respectively. The difference between $T = 1$ and $T = 5$ was no more than 8%, 30%, and 49% for the three clusterings.

Figure 6.5 shows the number of executions reviewed for different values of $T$. As with the number of failures discovered (Figure 6.2); the number of executions affected by the threshold increases as the number of clusters increases. While using a high value for $T$ along with a high number of clusters allows developers to discover a high number of failures, it comes at a price: With $k$ equal to 30% of the total number of executions, CBF with $T = 5$ requires developers to review roughly half of all the executions, regardless of label (48.55%, 62.02%, 60.99%, 64.00%, 62.24%, 57.74% for GCC, Javac, JTidy, ROME-F, ROME-L, and Xerces respectively). This is in addition to any other executions that CBF will label for review.

Figure 6.6 shows the % mean executions reviewed for RAF and CBF with all three clusterings ($T = 1$). RAF+ is not shown as it reviews the exact same number of executions as CBF. Clearly, part of CBF’s ability to discover more failures than RAF is that it reviews more executions: with $T = 1$, CBF reviews as much as 16.47% more executions than RAF (and higher for higher values of $T$). RAF+, however, illustrates that simply reviewing more executions does not reveal additional failures or defects (see Figure 6.3).

Table 6.4 shows the percentage increase between RAF and CBF for both $R$ and $F_d$. For example, CBF with 10% clustering and $T = 1$ reviews 15.14% more executions than RAF and discovers 20.01% more failures. At the very least, reviewing $X$% more
Figure 6.5: Executions Reviewed for differing $T$ values
Figure 6.6: % Mean Executions Reviewed vs $p_{ml}$ for CBF and RAF
executions with CBF yields at least X% more failures. It should be noted that these percentages require reviewing a number of executions for each discovered failure. In the above example (GCC, 10% clustering, T = 1), CBF reviews 504 more executions to discover 27.2 more failures. As noted in Section 6.1, CBF performed best on the Xerces dataset. According to Table 6.4 CBF reviewed 2.06% more executions than RAF and discovered 29.46% more failures. This translates into CBF reviewing 103 more executions and discovering 73.65 more failures.

The number of executions developers must review to discover a failure rises as the $p_{ml}$ increases. The rate at which this metric increases is different depending on the dataset. Figure 6.7 shows the number of executions reviewed per failure discovered for all six datasets. For both GCC and Javac, CBF lead developers to review fewer executions per failure than RAF when $p_{ml} \geq 0.26$ for GCC (9.07 vs. 9.20 with the 10% clustering) and Javac (5.16 vs. 5.36 with the 20% clustering). With JTidy, CBF with both the 10% and 20% clusterings reviewed fewer executions per failure than RAF once $p_{ml} \geq 0.20$ (6.95 and 7.17 executions for the 10% and 20% clusterings, and 7.23 for RAF). ROME-F was the only dataset where RAF always reviewed fewer executions per failure discovered than CBF. With ROME-L, CBF with the 20% clustering lead developers to review fewer executions than RAF with $p_{ml} \geq 0.2$ (5.58 executions vs. 5.74 for RAF). With Xerces, CBF with 10% clustering reviews

<table>
<thead>
<tr>
<th>Program</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$R$</td>
<td>$F_d$</td>
<td>$R$</td>
</tr>
<tr>
<td>GCC</td>
<td>15.14%</td>
<td>20.01%</td>
<td>7.12%</td>
</tr>
<tr>
<td>Javac</td>
<td>12.74%</td>
<td>24.71%</td>
<td>7.25%</td>
</tr>
<tr>
<td>JTidy</td>
<td>3.75%</td>
<td>27.06%</td>
<td>4.56%</td>
</tr>
<tr>
<td>ROME-F 5%</td>
<td>16.21%</td>
<td>18.61%</td>
<td>10.22%</td>
</tr>
<tr>
<td>ROME-L 5%</td>
<td>5.25%</td>
<td>26.61%</td>
<td>4.27%</td>
</tr>
<tr>
<td>Xerces 5%</td>
<td>2.06%</td>
<td>29.46%</td>
<td>4.52%</td>
</tr>
</tbody>
</table>
fewer executions per failure than RAF starting with $p_{ml} = 0.10$. As the mislabeling rate increases above these thresholds, the gap between CBF and RAF increases. In fact, for JTidy, ROME-L, and Xerces, all three clusterings (10%, 20%, and 30%) out-perform RAF with $p_{ml} = 0.30$. For GCC and Javac, both the 10% and 20% clusterings out-perform RAF.

### 6.4 CBF with $T = 0$

The results in the three previous sections (6.1, 6.2, and 6.3) illustrate that higher values of $T$ (2, 3, 4, 5) do not help discover failures any faster than $T = 1$. In fact, $T = 1$ requires the fewest reviews per failure discovered than all higher $T$ values. As the number of clusters increases, this disparity grows.

This raises the issue of whether analysis of SMALL clusters is at all beneficial. In this section, we compare CBF with $T = 1$ (the lowest value of $T$ considered, thus far) with RAF and CBF with $T = 0$. With $T = 0$, there are no SMALL clusters; instead, all of the clusters are initially labeled either SUSPECT or OK, and finally labeled BAD or OK.

Section 6.4.1 compares the number of failures discovered by CBF with $T = 0$ and $T = 1$, Section 6.4.2 looks at the number of defects discovered by the two methods, and Section 6.4.3 looks at the number of executions reviewed by the two techniques as well how many executions are reviewed per failure discovered.

#### 6.4.1 Failures Discovered

CBF with $T = 0$ allowed developers to discover nearly as many failures as CBF with $T = 1$ for all six datasets and across all three clusterings.

Given the mislabeling rate, $p_{ml}$, we can compute the difference between $T = 1$ and $T = 0$. If $NCF_1$ is the number of singleton clusters with a failure, then the difference
Figure 6.7: Executions reviewed per failure discovered vs $p_{ml}$ for CBF and RAF
in $F_d$ between CBF with $T = 0$ and $T = 1$ should be described by $(1 - p_{ml})NCF_1$. According to Figure 6.2, the number of singleton clusters increases as the number of clusters increases. The difference in $F_d$ between $T = 1$ and $T = 0$ should increase with the number of clusters. Figure 6.8 shows the percent failures discovered for CBF0, CBF1, and RAF for all three clusterings (10%, 20%, and 30%).

- For GCC, the difference in $F_d$ for $T = 1$ and $T = 0$ is less than 1.67% (2.27 failures) for all three clusterings.

- For Javac, the difference in $F_d$ for $T = 1$ and $T = 0$ is 0.02%, 1.27%, and 3.44% for the 10%, 20%, and 30% clusterings, respectively. This is equivalent to 0.05, 2.97, and 8.01 failures.

- JTidy saw the largest increase between clusterings. For the 10%, 20%, and 30% clusterings, the difference between $T = 0$ and $T = 1$ for $F_d$ was 0.55% (1.7 failures), 2.72% (8.37 failures), and 5.53% (17.02 failures).

- For ROME-F, CBF with $T = 0$ and $T = 1$ differed by 0.41% (1.02 failures) and 0.77% (1.92 failures) for the 10% and 20% clusterings. For the 30% clustering, they differed by 2.12% (5.29 failures). On the other hand, for ROME-L, $T = 1$ discovers 0.24% (0.59), 1.25% (3.12 failures), and 3.60% (8.99 failures) more failures than $T = 0$.

- For Xerces, there was less than a 1% (2.51 failures) difference in $F_d$ between $T = 0$ and $T = 1$ for all clusterings.

In terms of $F_d$, CBF with $T = 0$ is useful alternative to CBF with $T = 1$, especially when the number of clusters is low (for example, the 10% clustering). With more clusters, CBF with $T = 0$ will still find many of the failures even with high mislabeling ($p_{ml} = 0.3$).
<table>
<thead>
<tr>
<th>Method</th>
<th>CBF0</th>
<th>CBF1</th>
<th>RAF</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Javac</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jtidy</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROME−F 5%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROME−L 5%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xerces 5%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.8: % Mean Failures Discovered vs $p_{ml}$ for CBF0, CBF1, and RAF
6.4.2 Defects Discovered

CBF with $T = 0$ caused developers to find fewer defects than $T = 1$ for all datasets. For two of the datasets, $T = 0$ revealed more than 5% fewer defects than $T = 1$ with the 30% clustering.

Table 6.5 shows the percentage defects discovered by CBF with $T = 0$ and $T = 1$ at $p_{ml} = 0.3$. CBF with $T = 1$ and $T = 0$ differ by less 1% for all three clusterings for GCC, JTidy, ROME-F, and Xerces. For Javac, $T = 1$ finds 0.72%, 2.28%, and 5.55% more defects than $T = 0$ for the 10%, 20%, and 30% clusterings, respectively. For ROME-L, CBF with $T = 1$ finds 1.14%, 3.57%, and 6.14% more defects than $T = 0$ for the three clusterings.

<table>
<thead>
<tr>
<th>Program</th>
<th>(T=1) 10%</th>
<th>(T=1) 20%</th>
<th>(T=1) 30%</th>
<th>(T=0) 10%</th>
<th>(T=0) 20%</th>
<th>(T=0) 30%</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>98.23</td>
<td>98.54</td>
<td>98.00</td>
<td>98.62</td>
<td>97.81</td>
<td>97.08</td>
</tr>
<tr>
<td>Javac</td>
<td>88.24</td>
<td>90.01</td>
<td>89.81</td>
<td>87.52</td>
<td>87.73</td>
<td>84.25</td>
</tr>
<tr>
<td>JTidy</td>
<td>98.88</td>
<td>99.38</td>
<td>98.38</td>
<td>99.00</td>
<td>99.25</td>
<td>98.75</td>
</tr>
<tr>
<td>ROME-F 5%</td>
<td>96.43</td>
<td>96.29</td>
<td>92.57</td>
<td>97.00</td>
<td>95.71</td>
<td>92.14</td>
</tr>
<tr>
<td>ROME-L 5%</td>
<td>92.00</td>
<td>95.29</td>
<td>95.86</td>
<td>90.86</td>
<td>91.71</td>
<td>89.71</td>
</tr>
<tr>
<td>Xerces 5%</td>
<td>98.82</td>
<td>99.41</td>
<td>99.41</td>
<td>99.18</td>
<td>98.88</td>
<td>98.59</td>
</tr>
</tbody>
</table>

CBF with $T = 1$ finds more defects, especially with more clusters (30% clustering), than CBF with $T = 0$. This is most apparent with the Javac dataset, where 26 of the confirmed failures are in singleton clusters: developers miss discovering 3.72 defects if they use CBF with $T = 0$. CBF with $T = 0$ should not be used if the application contains many rare, expensive defects: these are defects that are costly for users but are difficult to find because they occur rarely. With these defects, developers may want to use CBF with $T \geq 1$. 

62
6.4.3 Executions Reviewed

The number of executions reviewed \((R)\) follows as a similar trend to the number of failures and defects discovered: as the number of clusters increases, the difference between \(T = 0\) and \(T = 1\) increases. CBF with \(T = 0\) allows developers to review fewer executions per failure discovered than \(T = 1\).

This same trend is seen when examining the number of executions reviewed \((R)\) by CBF with \(T = 0\) and \(T = 1\). Similar to the number of failures discovered, the difference in \(R\) between \(T = 0\) and \(T = 1\) is directly related to the number of singleton clusters. If \(NC_1\) is the number of clusters with \(|C| = 1\) (note that \(NC_1 \geq NCF_1\)), then the difference between CBF with \(T = 0\) and \(T = 1\) is \((1-p_{ml})NCF_1 + p_{ml}(NC_1 - NCF_1)\) executions reviewed. Figure 6.9 shows the number of executions reviewed for the different \(p_{ml}\) values and clusterings. In particular, for \(p_{ml} = 0.3\):

- For \(GCC\), \(R\) for \(T = 0\) and \(T = 1\) differs by 0.10\%, 0.53\%, and 2.15\% for the 10\%, 20\%, and 30\% clustering. This amounts to 3.44, 17.65, and 71.62 reviews.

- For \(Javac\), \(T = 0\) and \(T = 1\) differ by 0.06\% (1.73 executions), and 1.81\% (56.88), and 5.05\% (158.64), for the three clusterings.

- For \(JTidy\), CBF with \(T = 0\) reviewed 0.68\% (54.59), 2.73\% (217.97), and 6.25\% (499.59) fewer executions than \(T = 1\) for the 10\%, 20\%, and 30\% clusterings, respectively.

- For \(ROME-F\), CBF with \(T = 0\) reviewed 0.07\% (3.47), 1.43\% (71.59), and 4.81\% (240.36) fewer executions than \(T = 1\) for the 10\%, 20\%, and 30\% clusterings. For \(ROME-L\), CBF with \(T = 0\) saved 0.38\% (19.33), 2.23\% (111.41), and 5.27\% (263.63) reviews over \(T = 1\) for the three clusterings.

- Finally, for \(Xerces\), CBF with \(T = 0\) reviewed 0.71\% (35.44), 3.19\% (159.44), and 6.82\% (341.19) fewer executions than \(T = 1\).
When the number of clusters is 10% of the total executions, developers will review similar numbers of executions when using CBF with $T = 1$ and $T = 0$. With more clusters, this gap widens. In particular, with the $JTidy$ dataset, developers review 499.59 more executions with $T = 1$ and the 30% clustering. This would result in 17.02 more failures (from Section 6.4.1), and no additional defects (from Section 6.4.2). CBF with $T = 1$ requires considerably more work than $T = 0$ when using the 20% and 30% clusterings.

Figure 6.10 shows the number of executions reviewed per failure discovered ($R/F_d$) for CBF with $T = 1$ and $T = 0$, as well as RAF. In general, CBF with $T = 1$ required developers to review more executions per failure discovered than CBF with $T = 0$. With $p_{ml} = 0.3$:

- For $GCC$, CBF with $T = 1$ reviewed 0.2, 0.15, and 0.42 more executions per failure discovered than CBF with $T = 0$, for 10%, 20%, and 30% clusterings, respectively. CBF with the 30% clustering and $T = 0$ was most efficient with 9.85 reviews per failure discovered.

- For $Javac$, CBF with $T = 1$ and $T = 0$ differed by 0.01, 0.18, and 0.53 executions per failure discovered for the three clusterings. CBF with the 30% clustering and $T = 0$ was most efficient with 5.41 reviews per failure discovered.

- $JTidy$ had one of the largest differences between $T = 0$ and $T = 1$. For the 10%, 20%, and 30% clusterings, CBF with $T = 1$ required developers to review 0.13, 0.47, and 1.15 more executions per failure discovered. CBF with the 20% clustering and $T = 0$ reviewed the fewest executions per failure discovered: 9.21.

- For $ROME-F$ and $ROME-L$, developers reviewed 0.06 more executions per failure discovered with the 10% clustering. For the 20% clustering, developers reviewed 0.25 and 0.37 more executions per failure using $T = 1$ instead of $T = 0$. For the 30% clustering, $T = 1$ and $T = 0$ differed by 0.9 and 0.82 $R/F_d$ for
Figure 6.9: % Mean Executions Reviewed vs $p_{ml}$ for CBF0, CBF1, and RAF
ROME-F and ROME-L, respectively. Overall, CBF with line coverage profiling (ROME-L), 20% clustering, and $T = 0$ was the most efficient technique for the ROME dataset with 7.16 reviews per failure discovered.

- Xerces exhibited the largest difference between $T = 0$ and $T = 1$ for all three clusterings. For the 10%, 20%, and 30% clusterings, CBF with $T = 1$ reviewed 0.16, 0.66, and 1.32 more executions per failure than $T = 0$. The most efficient method was CBF with 10% clustering and $T = 0$: it reviewed 6.69 executions per failure discovered.

CBF with $T = 0$ represents an improvement over CBF with $T = 1$ at all three clusterings. It allows developers to review fewer executions to discover similar numbers of failures. Using $T = 0$ can, however, cause developers to miss unusual defects (defects only present in singleton clusters). The need to discover all defects in an application must therefore be weighed against the increased work required by CBF with $T \geq 1$. 

66
Figure 6.10: Executions reviewed per failure discovered vs $p_{ml}$ for CBF0, CBF1, and RAF
Chapter 7

Evaluation of Review-All-FAILUREs and \textit{k}-Nearest Neighbors

This chapter describes the results of the empirical study conducted for RAF+$k$NN, which is described in Section 5.4. Section 7.1 explains which variants of CBF were used in the experiments. Section 7.2 compares the numbers of failures discovered for CBF, RAF+$k$NN, and RAF. Section 7.3 shows the number of defects discovered by each method. Finally, Section 7.4 compares the amount of work (measured by executions reviewed) developers must do under each method.

7.1 Comparing CBF and RAF+$k$NN

Since CBF was evaluated in detail in Chapter 6, only results for the “best” value of the number of clusters $NC$ are considered here. The “best” value of $NC$ was chosen by picking the value for $NC$ that minimizes the number of executions reviewed per failure discovered when $p_{mf} = 0.3$ and $T = 1$. This ratio ($\frac{R}{F_d}$) is shown in Table 7.1. The $NC$ values used in this study are found in column 2. In this chapter, we compare
RAF+kNN to RAF, and CBF with $T = 1$ (denoted “CBF1”) and $T = 0$ (denoted “CBF0”).

### 7.2 Failures Discovered

RAF+kNN discovered fewer failures than CBF, but more than RAF. At worst (with $k = 1$), it discovers 10%-15% more failures than RAF. At best (with $k = 5$), it discovers 2% fewer failures than CBF0.

Figure 7.1 shows the percentage failures discovered for both mislabeling models. The different mislabeling models had no effect on the number of failures discovered.

- With $GCC$, RAF+kNN discovered 80.11%, 83.37%, and 86.76% of failures with $k = \{1, 3, 5\}$. This is at least 9.68% more than RAF, and at worst 6.82% less than CBF0.

- With $Javac$, RAF+kNN discovered 85.26%, 92.28%, and 93.88% of failures with $k = \{1, 3, 5\}$. This is at least 14.96% more than RAF and, with RAF+5NN, 0.05% more than CBF0.

- With $JTidy$, RAF+kNN discovered 84.24%, 92.03%, and 94.10% of failures with $k = \{1, 3, 5\}$, respectively. This represents an improvement of at least 14.61%
Figure 7.1: % Mean Failures Discovered vs. $p_{mf}$
compared with RAF. Compared with CBF0, RAF+kNN found between 2.04% and 11.90% fewer failures.

- With *ROME-F*, RAF+kNN discovered 75.92%, 81.01%, and 83.90% of failures with $k = \{1, 3, 5\}$, respectively. This is an improvement of at least 5.26% over RAF. RAF+kNN finds between 2.04% and 10.02% fewer failures than CBF0. With *ROME-L*, on the other hand, RAF+kNN discovered 85.35%, 93.25%, and 94.74% of failures with $k = \{1, 3, 5\}$. This was at least 14.73% more than RAF and between 0.44%-9.84% less than CBF0.

- With *Xerces*, RAF+kNN found 86.19%, 95.22%, and 97.85% of failures. This is at least 15.98% more than RAF, and between 2.13% and 13.79% less than CBF0.

Overall, RAF+1NN performs very favorably when compared with RAF. It reviews fewer executions than CBF, but discovers at least 10% to 15% more failures (the lone exception being *ROME-F*, where RAF+1NN finds only 75.92% of failures).

RAF+3NN represents a modest improvement over RAF+1NN. For *Javac*, *JTidy*, *ROME-L*, and *Xerces* it finds a minimum of 22% more failures than RAF. For *GCC* and *ROME-F*, RAF+3NN finds 12.93% and 10.35% more failures.

For most of the data sets, RAF+3NN discovers 7% or more failures than RAF+1NN. RAF+5NN, on the other hand, finds only 1.5%-3.4% more failures than RAF+3NN.

Differential mislabeling has no effect (relative to uniform mislabeling) on the number of failures discovered. In order for a developer to correctly identify an execution as a failure, they must either review an execution correctly labeled FAILURE \(((1 - p_{mf}) \times F_a)\), or they must review an execution labeled SUCCESS \(p_{mf} \times F_a\). The number of mislabeled successes has no bearing on this metric.
7.3 Defects Discovered

Like the number of failures discovered, RAF+$k$NN does not tend to find as many defects as CBF1. With $k = 3$ or $k = 5$, it did, however, find approximately as many defects as CBF0. RAF+$k$NN found more defects than RAF.

Figure 7.2 gives the defects discovered for each technique and data set with $p_{mf} = 0.3$. CBF1 finds the most defects, while RAF+$k$NN finds either as many or more defects than RAF. RAF+$k$NN found at least 89.71% of failures on each data set except Javac. For Xerces, ROME-L, and JTidy, RAF+$3$NN found more defects than RAF+$1$NN and at least as many defects as RAF+$5$NN.

- With GCC, RAF+$k$NN with $k = \{1, 3, 5\}$ discovered 96.65%, 96.62%, and 97.12% of all the defects.

- With Javac, RAF+$k$NN discovered 84.70%, 86.60%, and 86.63% of the defects with $k = \{1, 3, 5\}$, respectively. This is 1.46%-3.39% more than RAF, 1.1%-3.03% less than CBF0, and 3.39%-5.31% less than CBF1 (90.01%).

- With JTidy, RAF+$k$NN discovered 96.63% of defects with $k = 1$ and 99% with $k = \{3, 5\}$. RAF+$3$NN and RAF+$5$NN find as many defects as CBF0 and 2.5% more defects than RAF.

- With ROME-F, RAF+$k$NN finds 91.43%, 90.71%, and 93% of the defects with $k = \{1, 3, 5\}$, respectively. This is 3.29%-5.57% less than CBF1 and 2.71%-5% less than CBF0. RAF+$k$NN improves on RAF by 0.86%-4.29%. With ROME-L, on the other hand, RAF+$k$NN discovers 90.29%, 91.14%, and 89.71% of the defects. This is between 4.14%-5.57% less than CBF1, 0.57%-2% less than CBF1, and similar to RAF.

- With Xerces, RAF+$k$NN discovered 97.06%, 98.12%, and 97.76% of the defects with $k = \{1, 3, 5\}$. This is no more than 2.12% less than CBF and no less than
Figure 7.2: % Mean Defects Discovered vs. $p_{mf}$
4.29% more than RAF.

Differential mislabeling has no effect the number of defects discovered. As with the number of failures discovered, the number of defects found by all of the techniques is not affected by making \( p_{ms} = 0.1 \): instead, as \( p_{mf} \) increases, the number of defects immediately discoverable by RAF (and therefore by RAF+\( k \)NN and CBF) decreases. RAF+\( k \)NN and CBF tend to discover more defects than RAF because they also review mislabeled failures.

7.4 Executions Reviewed

Figure 7.3 shows the plots of executions reviewed for each technique and data set under both mislabeling models. Across all data sets, techniques, and mislabeling models, RAF+\( k \)NN requires reviewing fewer executions than CBF but more than RAF. Under uniform mislabeling, the average number of executions reviewed increases linearly with \( p_{mf} \).

With the exception of GCC, CBF1 required reviewing more executions than both CBF0 and RAF+5NN. For GCC, RAF+5NN prompted developers to review 1.03% more executions than CBF1. With 30% clustering, the GCC data set was partitioned into 999 clusters. The mean size for a cluster was 3.34. Because the majority of clusters had fewer than five elements (81% of the clusters have \( |C| < 5 \)), RAF+5NN tended to review more executions per discovered failure than CBF in this instance. For Javac, ROME-L, and Xerces, RAF+5NN required reviewing 1.53%, .58%, and .02% more executions than CBF0 and 0.28%, 1.35%, and 0.69% fewer than CBF1. For JTidy and ROME-F, RAF+5NN reviewed 0.68% and 1.44% fewer executions than CBF0 and 2.04% and 4.31% fewer executions than CBF1. Because of this, RAF+5NN does not provide any noticeable advantage over CBF at \( p_{mf} = 0.3 \): with or without corroboration, it reviews a similar number of executions but finds fewer
RAF+3NN prompted developers to review 0.65%, 1.39%, 3.95%, and 0.23% fewer executions than CBF0 for JavaC, JTidy, ROME-F, and Xerces at $p_{mf} = 0.3$. This is, respectively, equivalent to 20.46, 111.17, 197.25, and 11.5 executions. This was particularly advantageous for JavaC, JTidy, and ROME-F, as only 3.62, 12.65, and 12.31 fewer failures were discovered. For Xerces, on the other hand, developers would have discovered 11.89 fewer failures with RAF+3NN; indicating that all of the executions developers did not review were failures. Finally, RAF+3NN required developers to review 0.77% (25.52 executions) and .07% (3.46 executions) more executions than CBF0 for GCC and ROME-L, respectively. In both cases, CBF0 still discovered more failures: 4.85 failures for GCC and 4.84 for ROME-L.

At $p_{mf} = 0.3$, RAF+1NN required developers to review 70.56 (2.12%), 110.66 (3.52%), 182.62 (2.29%), 361.43 (7.23%), 59.73 (1.19%), and 37.85% (0.76%) fewer executions than CBF for GCC, JavaC, JTidy, ROME-F, ROME-L, and Xerces, respectively.

Xerces exhibits the smallest differences between RAF, CBF, and RAF+$k$NN: The difference between CBF and RAF is only 103.23 executions (2.06% of all Xerces’ executions). ROME-F has the largest difference among the techniques: CBF reviews 511.21 (10.22%) more executions than RAF. For the remaining data sets, the maximum difference between RAF and the most expensive technique is between 3.76% and 7.25%.

Section 7.2, demonstrates that both RAF+1NN and RAF+3NN find significantly more failures than RAF. In addition, over all data sets, RAF+1NN reviews less than 100 extra executions over RAF at $p_{mf} = 0.3$.

The stark difference between the uniform and differential mislabeling strategies is due to the types of executions that get mislabeled. With RAF, only executions labeled FAILURE are reviewed. Given the number of actual failures and successes ($F_a$ and
Figure 7.3: % Mean Executions Reviewed vs. $p_{mf}$
we can calculate the number of executions reviewed by RAF using equation 7.1.

\[
R = (1 - p_{mf}) \times F_a + p_{ms} \times S_a \tag{7.1}
\]

Under differential mislabeling, \( R \) decreased slightly (at most 3.56\%) as \( p_{mf} \) increased from 0 to 0.3. The relationship between the techniques stays the same under differential mislabeling: CBF reviews the most executions, followed by RAF+5NN, RAF+3NN, and RAF+1NN. RAF always reviews the fewest executions. At \( p_{mf} = 0.0 \), All of the techniques review between 7\% and 9.5\% fewer executions under uniform mislabeling than they would under differential mislabeling. When \( p_{mf} \) rises to 0.3, uniform mislabeling causes each technique to review between 16\% and 19\% more executions than under differential mislabeling.

Under differential mislabeling, the value of \( p_{ms} \times S_a \) is fixed (.1 \( \times S_a \)). \( R \) is therefore dependent on the probability \((1 - p_{mf}) \times F_a\), which decreases as the mislabeling rate increases. More mislabeled failures mean fewer failures for developers to review. For uniform mislabeling \( p_{mf} = p_{ms} \), thus the number of actual failures reviewed decreases and the number of mislabeled successes increases as \( p_{mf} \) and \( p_{ms} \) increase. Because all of the data sets have far more successes than failures, there is a net increase in \( R \).

Figure 7.4 plots the number of executions reviewed per failures discovered (\( R/F_d \)) for the different values of \( p_{mf} \) (for both uniform and differential mislabeling). For low mislabeling (where \( p_{mf} \leq .1 \)), RAF tended to have the lowest \( R/F_d \). When \( 0.10 \leq p_{mf} \leq 0.20 \), RAF+1NN or RAF+3NN tended to have the lowest \( R/F_d \). For high mislabeling (\( p_{mf} \leq 0.26 \)), most of the datasets benefited from CBF0. CBF1 was less efficient than both CBF0 and RAF+\( k \)NN for at least one \( k \) value across all \( p_{mf} \) values and data sets.

- For GCC, RAF had the lowest \( R/F_d \) when \( p_{mf} < 0.18 \). With \( 0.18 \leq p_{mf} < 0.26 \), RAF+1NN was the most efficient. With \( p_{mf} > 0.26 \), CBF0 was the most
Figure 7.4: Executions reviewed / Failure Discovered vs. $p_{mf}$
efficient method.

- For *Javac*, RAF was the most efficient with \( p_{mf} < 0.14 \). RAF+1NN had the lowest \( R/F_d \) when \( 0.14 \leq p_{mf} < 0.30 \). However, at \( p_{mf} = 0.30 \), CBF0, RAF+1NN and RAF+3NN all had the lowest \( R/F_d = 5.5 \) executions per failure.

- For *JTidy*, RAF produced the lowest \( R/F_d \) while \( p_{mf} < 0.16 \). With \( 0.16 \leq p_{mf} < 0.20 \), RAF+1NN was most efficient. With \( 0.20 \leq p_{mf} < 0.30 \), RAF+3NN had the lowest \( R/F_d \). Like *Javac*, CBF0 was the most efficient when \( p_{mf} = 0.30 \).

- The results from *ROME-F* reflect the poor choice in profiling: RAF produced the lowest \( R/F_d \) values for \( p_{mf} < 0.26 \). With \( p_{mf} \geq 0.26 \), RAF+1NN was most efficient.

- RAF+\( k \)NN and CBF performed much better on *ROME-L*, again demonstrating the effects of a good choice of profiling technique. RAF had the lowest \( R/F_d \) when \( p_{mf} < 0.10 \). With \( 0.10 \leq p_{mf} < 0.16 \), RAF+1NN had the lowest \( R/F_d \). With \( p_{mf} \geq 0.16 \), CBF0 was again the most efficient technique.

- CBF0 was consistently the best method for the *Xerces* dataset — CBF0 had the lowest \( R/F_d \) for \( p_{mf} \geq 0.04 \), though both RAF+3NN and RAF+5NN had fairly close values (within .02 executions per failure).

The results were very different for differential mislabeling. Excepting *ROME-L* and *Xerces*, either RAF or RAF+1NN tended to be the most efficient methods, even for higher values of \( p_{mf} \).

- With *GCC*, RAF produced the lowest \( R/F_d \) values for \( p_{mf} < 0.26 \). For \( p_{mf} \geq 0.26 \), RAF+1NN was the most efficient.

- The results were similar for *Javac*: for \( p_{mf} < 0.14 \), RAF was the most efficient. For \( p_{mf} \geq 0.14 \), RAF+1NN had the lowest \( R/F_d \).
• For *JTidy*, RAF had the lowest $R/F_d$ for $p_{mf} < 0.08$. For $0.08 \leq p_{mf} < 0.28$, RAF+1NN was most efficient. Finally, above $p_{mf} = 0.28$, RAF+3NN had the lowest $R/F_d$.

• The results for *ROME-F* were similar to the uniform mislabeling results. RAF was the most efficient method across all $p_{mf}$ values.

• For *ROME-L*, on the other hand, RAF was most efficient only up to $p_{mf} < 0.08$. From $0.08 \leq p_{mf} < 0.26$, RAF+1NN produced the lowest $R/F_d$ values. With $p_{mf} \geq 0.26$, CBF0 was best.

• Unlike the rest of the datasets, the best method for *Xerces* was CBF0 for all $p_{mf}$ values, though, for smaller $p_{mf}$ values, RAF+3NN was very close (within .02 executions per failure).

In general, this study demonstrated several facts: CBF0 (with zero threshold) tended to be the best technique when the dataset had high uniform mislabeling; indicating that CBF is very good at avoiding mislabeled successes. RAF+$k$NN, on the other hand, performed very well with $k = \{1, 3\}$ for “midrange” uniform mislabeling values (roughly $0.14 < p_{mf} < 0.24$). With differential mislabeling, RAF+$k$NN tended to outperform both RAF and CBF0 for mid to high $p_{mf}$ values. RAF was best for low $p_{mf}$ values under both mislabeling models.

In terms of finding defects, CBF1 found the most defects, followed by CBF0 and RAF+$k$NN. The differences between the five methods is slight, however (within 10% for all of the datasets with $p_{mf} = 0.3$). The trade off between the increased executions reviewed and defects discovered is explored in greater detail in the next chapter (Chapter 8).
Chapter 8

Analysis of Relative Costs of CBF, RAF+$k$NN, and RAF

How much effort a software development organization is willing to spend reviewing executions in order to identify actual failures depends on both the cost of reviewing executions and the cost of overlooking defects (so that they remain in deployed software). Though these costs are application dependent, it is informative to compare our techniques under simplifying assumptions about the relative magnitudes of these costs. We represent the immediate cost to an organization of applying one of our methods to an application by the number $R_u$ of “review cost units” this entails. One review cost unit corresponds to the average cost of developers reviewing 100 executions. (Again, we assume perfect debugging.) To allow an organization to examine the trade off between reviewing more executions and overlooking defects, we introduce a multiplier $M$ that indicates how many review cost units an overlooked defect is “worth” on average, in terms of total long-term cost to the organization due to lost customers, decreased sales, legal liability, etc. Let $D$ be the number of defects remaining after the application of a particular technique. The total long-term cost of
If all other factors are fixed, an organization should prefer to use the technique with the lowest value of $C$.

Figure 8.1 shows the effects of $M$ on $C$ for RAF, RAF+$k$NN (with $k = 1, 3, 5$), and CBF with $p_{mf} = \{0.1, 0.2, 0.3\}$ for GCC under both uniform and differential mislabeling. The lowest cost method is dependent on two factors: the mislabeling rate and the $M$: under low mislabeling ($p_{ml} = 0.1$), RAF is the least expensive method when missed defects are inexpensive: With $M = 10$, RAF costs 5.54, RAF+3NN costs 7.54, and CBF costs 8.12. As $M$ increases, however ($M = 50$), the order reverses: RAF costs 13.42, RAF+3NN costs 11.87, and CBF costs 12.22. As $p_{mf}$ increases, each method becomes more expensive as $M$ increases. With $p_{mf} = 0.3$, RAF increases from 19.72

\[ C = R_u + M \times D \quad (8.1) \]
to 49.01 and RAF+5NN increases from 22.04 to 48.2 as $M$ goes from 10 → 50. The cost of CBF increases from 16.87 → 34.37. The relatively small differences in cost between all the methods reflects the small differences in defects discovered for GCC seen in figure 7.2. Under high mislabeling with very expensive defects, CBF tends to be the least expensive method for GCC.

$Javac$, shown in figure 8.2, provides a much clearer separation between methods. RAF is consistently among the most expensive methods. Under low mislabeling and $M$ ($p_{mf} = 0.1$ and $M = 10$), RAF is 1.11, 1.14, and 1.39 times more expensive than RAF+3NN, RAF+5NN, and CBF, respectively. Under the same mislabeling but with $M = 50$, RAF is 1.18, 1.34, and 1.78 times more expensive than RAF+3NN, RAF+5NN, and CBF. These trends hold true across mislabeling rates and $M$ values. CBF is the most desirable method, followed by RAF+$k$NN (with higher $k$ values being better), and RAF.
Figure 8.3: *JTidy*: Total Cost vs. Multiplier

Figure 8.3 shows the total cost for *JTidy*. As the mislabeling rate increases from 0.1 to 0.3 and $M$ increases from 10 to 50, the different methods divide into two different sets. RAF and RAF+1NN are the more expensive methods, and RAF+3NN, RAF+5NN, and CBF are the less expensive methods. With $M = 10$ and $p_{mf} = 0.1$, RAF+$k$NN and CBF are as costly as RAF. For higher $M$ values, RAF+3NN is the least expensive method: RAF is 1.09 and 1.18 times more expensive than RAF+3NN when $M = 30$ and $M = 50$. CBF is 1.20 and 1.13 times more expensive than RAF+3NN. When $p_{mf} = 0.3$, these differences increase: At $M = 50$, RAF and CBF are 1.37 and 1.28 times as expensive as RAF+3NN.

*ROME-F* (figure 8.4) and *ROME-L* (figure 8.5) differ in how much the methods cost. With $p_{mf} = 0.1$, CBF is the most expensive method. CBF is between 1.39 and 1.69 times as expensive as RAF as $M$ decreases from 50 to 10. Under the same conditions, using line coverage (*ROME-L*), CBF goes from being 1.2 times as
Figure 8.4: *ROME-F 5*: Total Cost vs. Multiplier

Figure 8.5: *ROME-L 5*: Total Cost vs. Multiplier
expensive as RAF ($M = 10$) to costing 0.70 times as much as RAF ($M = 50$). With $p_{mf} = 0.1$, RAF+$k$NN with $ROME-F$ is more expensive than RAF (between 1.02 and 1.12) times as expensive with $k = 3$. Under $ROME-L$, however, RAF+$k$NN is either equivalent or 0.91 times less expensive than RAF.

Like Javac, Xerces provides a clear separation between the different methods. RAF is the most expensive method across all values of $p_{mf}$ and $M$. The relative costs for Xerces are shown in figure 8.6. With $p_{mf} = 0.1$ and $M = 30$, RAF+5NN is .45 times less expensive than RAF and .95 times less expensive than CBF. With $M = 50$, RAF+5NN is .30 times less expensive as RAF and .75 times less expensive than CBF. Under higher mislabeling rate ($p_{mf} = 0.3$) CBF becomes the least expensive method. With $p_{mf} = 0.3$ and $M=50$, CBF is .32, .94, and .68 times the cost of RAF, RAF+3NN, and RAF+5NN.

<table>
<thead>
<tr>
<th>Method</th>
<th>Uniform Mislabeling</th>
<th>Differential Mislabeling</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBF</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>RAF+1NN</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>RAF+3NN</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>RAF+5NN</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>RAF</td>
<td>□</td>
<td>□</td>
</tr>
</tbody>
</table>

![Figure 8.6: Xerces 5%: Total Cost vs. Multiplier](image)

86
8.1 Effects of Differential Mislabling on Relative Cost

The figures shown for relative cost show the results for both uniform and differential mislabeling. Differential mislabeling does not change the relationship between methods for each of the datasets. With higher values of $p_{mf}$, however, differential mislabeling tends to be less expensive than uniform mislabeling under the same conditions. The cost of each method is equivalent when $p_{mf} = 0.1$. This makes sense: $p_{mf} = p_{ms} = 0.1$. For many of the datasets, the method cost under uniform mislabeling rises as $p_{mf}$ increases. This is especially pronounced for $JTidy$ (figure 8.3) and $ROME-L$ (figure 8.5).
Chapter 9

Evaluation of Round-robin Cluster Sampling with \( k \)-NN Ranking

This chapter presents the results of the empirical study comparing round-robin cluster sampling with RAF, which is described in Section 5.6. Section 9.1 evaluates how quickly each method discovers actual failures, and Section 9.2 demonstrates how quickly each method discovers defects.

When comparing RRCS and RAF, it is useful to consider the percentage of failures discovered and the percentage of executions reviewed. These values vary depending on both the mislabeling rate and the mislabeling model. For example, the GCC data set has 136 actual failures. With \( p_{ml} = 0.2 \), only 108.8 of those failures will be

<table>
<thead>
<tr>
<th>Data Set</th>
<th>( p_{mf} ) = 0.15</th>
<th>( p_{mf} ) = 0.20</th>
<th>( p_{mf} ) = 0.25</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>115.6</td>
<td>108.8</td>
<td>102</td>
</tr>
<tr>
<td>Javac</td>
<td>198.05</td>
<td>186.4</td>
<td>174.75</td>
</tr>
<tr>
<td>JTidy</td>
<td>261.8</td>
<td>246.4</td>
<td>231</td>
</tr>
<tr>
<td>ROME</td>
<td>212.5</td>
<td>200</td>
<td>187.5</td>
</tr>
<tr>
<td>Xerces 5%</td>
<td>212.5</td>
<td>200</td>
<td>187.5</td>
</tr>
</tbody>
</table>
Table 9.2: Maximum Number of Executions Reviewed with RRCS for differing $p_{ms}$ Values

<table>
<thead>
<tr>
<th>Data Set</th>
<th>$p_{ms}$ 0.08</th>
<th>$p_{ms}$ 0.10</th>
<th>$p_{ms}$ 0.13</th>
<th>$p_{ms}$ 0.15</th>
<th>$p_{ms}$ 0.20</th>
<th>$p_{ms}$ 0.25</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>355.38</td>
<td>428.5</td>
<td>501.63</td>
<td>595.15</td>
<td>748.2</td>
<td>901.25</td>
</tr>
<tr>
<td>Javac</td>
<td>416.08</td>
<td>477.1</td>
<td>538.13</td>
<td>634.1</td>
<td>767.8</td>
<td>901.5</td>
</tr>
<tr>
<td>JTidy</td>
<td>837.95</td>
<td>1014.6</td>
<td>1191.25</td>
<td>1414.1</td>
<td>1782.8</td>
<td>2151.5</td>
</tr>
<tr>
<td>ROME</td>
<td>568.75</td>
<td>675</td>
<td>781.25</td>
<td>925</td>
<td>1150</td>
<td>1375</td>
</tr>
<tr>
<td>Xerces 5%</td>
<td>568.75</td>
<td>675</td>
<td>781.25</td>
<td>925</td>
<td>1150</td>
<td>1375</td>
</tr>
</tbody>
</table>

properly labeled FAILURE; thus, neither technique can discover more failures than this (since they do not review any executions labeled SUCCESS). Table 9.1 shows the available failures for different values of $p_{mf}$ and Table 9.2 shows the total executions reviewed for differing $p_{ms}$ values.

9.1 Failures Discovered

In general, RRCS requires fewer executions reviewed to discover each failure than does RAF. GCC, found in Figure 9.1, shows the mean number of executions that had to be reviewed to discover each actual failure for the GCC dataset using both RAF and RRCS. With uniform mislabeling and $p_{ml} = 0.15$ RAF requires developers to review 349.94 to discover 50% of the failures in GCC. Under the same conditions, RRCS reviews only 320.31. This difference does not change for higher $p_{ml}$ values; at 0.20, this difference is only 30.27 (4%). At $p_{ml} = 0.25$, RRCS reviews 38.02 (4.2%) fewer executions than RAF. To discover the majority of defects (100), the difference between RAF and RRCS narrows: RAF reviews 16.19, 14.96, and 9.26 more executions than RRCS for $p_{ml} = \{0.15, 0.20, 0.25\}$.

Under differential mislabeling, the situation stays roughly the same; both techniques review fewer executions. To find half of the failures, RRCS reviews roughly 20-30 fewer executions. To find 100 failures, RRCS reviews between 10 and 20 fewer
executions than RAF.

For *Javac*, shown in Figure 9.2, RRCS shows a greater improvement over RAF than for *GCC*. To find 50% of the failures present in *Javac* with uniform mislabeling, RRCS reviewed 25.94, 36.27, and 62.73 fewer executions than RAF with $p_{ml} = \{0.15, 0.20, 0.25\}$, respectively. To find all available failures, RRCS reviews 17.32, 32.2, and 59.93 fewer executions with the same mislabeling rates. With differential mislabeling, this difference narrows: RRCS saved only 20.22 reviews over RAF with $p_{mf} = 0.25$.

For *JTidy* (shown in Figure 9.3), RRCS reviews between 13%, 15.6%, and 16.8% fewer executions than RAF when discovering 50% of all failures under uniform mislabeling ($p_{ml} = \{0.15, 0.20, 0.25\}$). This lead narrows slightly when discovering all available failures: RRCS saves 9.8%, 11.8%, and 11.7% with the same $p_{ml}$ values. RRCS requires developers to review between 6-9% fewer executions than RAF to
discover all actual failures with differential mislabeling ($p_{mf} = \{0.15, 0.20, 0.25\}$).

Figure 9.4 shows the RRCS/RAF results for $ROME-F$ and $ROME-L$. RRCS with function-call coverage ($ROME-F$ 5\%) performs similarly to RAF: the executions reviewed values are within 1\% of RAF over all mislabeling rates and both mislabeling models. When using line coverage ($ROME-L$ 5\%), however, RRCS reviews 7\% fewer executions than RAF with $p_{ml} = \{0.20, 0.25\}$ and uniform mislabeling. Under differential mislabeling, RRCS with line-coverage saves 5.1\% and 5.8\% reviews with $p_{mf} = 0.20$ and 0.25.

With $Xerces$ (Figure 9.5), RRCS reviews 6.2\%, 8.3\%, and 10.7\% fewer executions than RAF when discovering all available failures under uniform mislabeling. Under differential mislabeling, this savings shrinks to 6\%, 5.3\%, and 6\%.
Figure 9.3: Mean Executions Reviewed vs. Failure Discovered for J* tidy

Figure 9.4: Mean Executions Reviewed vs. Failure Discovered for ROME
Table 9.3: Mean Executions Reviewed vs. Failure Discovered for $k = \{0, 1, 3\}$

<table>
<thead>
<tr>
<th>Program</th>
<th>$k$</th>
<th>Uniform 0.15</th>
<th>Uniform 0.20</th>
<th>Uniform 0.25</th>
<th>Differential 0.15</th>
<th>Differential 0.20</th>
<th>Differential 0.25</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0 578.03</td>
<td>713.14</td>
<td>856.02</td>
<td>339.21</td>
<td>405.32</td>
<td>475.57</td>
</tr>
<tr>
<td>GCC</td>
<td>1</td>
<td>558.06</td>
<td>692.23</td>
<td>833.93</td>
<td>334.49</td>
<td>401.33</td>
<td>461.88</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>561.54</td>
<td>697.88</td>
<td>838.76</td>
<td>332.95</td>
<td>396.43</td>
<td>473.75</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 601.98</td>
<td>708.48</td>
<td>799.27</td>
<td>401.32</td>
<td>455.45</td>
<td>495.29</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 582.84</td>
<td>689.89</td>
<td>781.85</td>
<td>396.93</td>
<td>439.28</td>
<td>480.36</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 589.25</td>
<td>684.48</td>
<td>787.08</td>
<td>394.27</td>
<td>437.28</td>
<td>483.65</td>
</tr>
<tr>
<td>Javac</td>
<td>0</td>
<td>1251.51</td>
<td>1546.66</td>
<td>1839.78</td>
<td>769.43</td>
<td>906.17</td>
<td>1052.04</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1240.65</td>
<td>1526.27</td>
<td>1795.48</td>
<td>762.38</td>
<td>902.27</td>
<td>1045.37</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1239.09</td>
<td>1528.3</td>
<td>1823.09</td>
<td>763.77</td>
<td>901.65</td>
<td>1026.96</td>
</tr>
<tr>
<td>JTidy</td>
<td>0</td>
<td>905.11</td>
<td>1115.59</td>
<td>1325.61</td>
<td>564.16</td>
<td>663.06</td>
<td>760.84</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>897.79</td>
<td>1104.43</td>
<td>1310.42</td>
<td>553.02</td>
<td>657.14</td>
<td>748.08</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>895.63</td>
<td>1107.27</td>
<td>1306.45</td>
<td>551.47</td>
<td>658.62</td>
<td>756.63</td>
</tr>
<tr>
<td>ROMF-5%</td>
<td>0</td>
<td>858</td>
<td>1043.74</td>
<td>1230.33</td>
<td>539.06</td>
<td>628.54</td>
<td>713.52</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>832.09</td>
<td>1035.2</td>
<td>1200.26</td>
<td>533.73</td>
<td>624.35</td>
<td>697.42</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>851.3</td>
<td>1026.86</td>
<td>1219.97</td>
<td>535.89</td>
<td>613.89</td>
<td>713.87</td>
</tr>
<tr>
<td>Xerces 5%</td>
<td>0</td>
<td>848.18</td>
<td>1025.98</td>
<td>1181.72</td>
<td>551.98</td>
<td>628.49</td>
<td>711.33</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>844.94</td>
<td>1011.45</td>
<td>1204.15</td>
<td>543.27</td>
<td>637.14</td>
<td>721.5</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>860.4</td>
<td>1008.66</td>
<td>1194.9</td>
<td>552.26</td>
<td>635.9</td>
<td>719.32</td>
</tr>
</tbody>
</table>
Table 9.3 gives the mean executions reviewed reviewed by RRCS with $k$-NN ranking to discover all correctly labeled failures under both uniform and differential mislabeling. With the exception of $ROME-L$ and $Xerces$, $k$-NN ranking reviews fewer executions than plain RRCS over all mislabeling rates and mislabeling models. Under uniform mislabeling with $p_{ml} = 0.25$ RRCS with $k$-NN ranking reviewed between 17.42 and 44.3 fewer executions than plain RRCS for $GCC$, $Javac$, and $ROME$. Using differential mislabeling ($p_{mf} = 0.25$), this improvement shrinks to between 12.76 and 25.08 executions reviewed. For $Xerces$, RRCS with $k$-NN ranking reviewed 22.43 ($k = 1$) and 13.8 ($k = 3$) more executions than plain RRCS with uniform mislabeling. With differential mislabeling, RRCS with $k$-NN ranking reviews 10.17 and 7.99 more executions than RAF with $k = 1$ and $k = 3$, respectively.

Figure 9.5: Mean Executions Reviewed vs. Failure Discovered for $Xerces$ 5%
For all mislabeling rates, data sets, and mislabeling models, RRCS required fewer reviewed executions to find a given number of defects when compared to RAF. In general, the first several defects are easily found by both techniques (requiring similar numbers of executions reviewed). Discovering the later defects (which tend to have fewer instances than earlier defects) tend to be significantly more efficient under RRCS than RAF.

Figure 9.6 shows the mean executions reviewed vs. defect discovered for GCC under both mislabeling models. For GCC, even discovering the first defect was faster via RRCS. Under uniform mislabeling, with \( p_{ml} = 0.15 \), RAF required an average of 5.51 reviews to discover the first defect. Under high mislabeling (\( p_{ml} = 0.25 \)), RAF required an average of 6.13 reviews to find the first defect. This is higher than RRCS, which reviews 1.87 and 2.22 executions under the same conditions. RRCS
did not fare as well on later defects. When discovering the last (26th) defect, RRCS reviews 5.67%, 2.62%, and 2.17% more FAILURE-labeled executions than RAF with $p_{ml} = \{0.15, 0.20, 0.25\}$. This same trend, where RRCS outperforms RAF at the beginning but not at end, is also seen in *Javac*. This is likely due to both *GCC* and *Javac* being synthetic test sets: *JTidy*, *ROME*, and *Xerces* all use operational inputs, which tend to have many executions for each defect and feature.

The effect of $k$-NN ranking on RRCS for *GCC* is shown in Figure 9.7. Both $k = 1$ and $k = 3$ provide a savings of 3% over RRCS without $k$-NN ranking with $p_{ml} = 0.15$. For higher $p_{ml}$ values, $k$-NN ranking provided no improvement over plain RRCS.

Differential mislabeling causes the number of executions reviewed for both RRCS and RAF to decrease. Under differential mislabeling, the number of FAILURE-labeled executions is essentially halved. The same trends seen with uniform mislabeling still hold: RRCS reviews fewer executions to find the first 18 defects. After that, RAF
reviews fewer executions to find the remaining 8 defects.

_Javac_ (shown in Figure 9.8) performed similarly to _GCC_. This is, again, likely due to both datasets being generated from synthetic test sets. With _Javac_, RAF required fewer reviewed executions to discover the first defect for all mislabeling rates (requiring 4.67 reviews at $p_{ml} = 0.25$, whereas RRCS required 5.52 reviews). The difference between RAF and RRCS rises steadily. To discover 50 defects, RAF requires an average of 549.13 reviews, compared with 448.7 reviews for RRCS (11% of all FAILUREs). To discover all defects, RRCS reviews 855.06 — 5% less than RAF.

When looking at differential mislabeling, both techniques once again have a much smaller pool of executions to review: at 25% mislabeling, uniform mislabeling gives 901 FAILUREs and differential mislabeling gives 538.13 FAILUREs.

Figure 9.9 shows the mean executions reviewed vs. defect discovered with differing $k$ values for _Javac_. $k = 3$ provides, at best, a 1% increase over plain RRCS.
Figure 9.9: Mean Executions Reviewed vs. Defects Discovered for Javac $k = \{0, 1, 3\}$

Figure 9.10: Mean Executions Reviewed vs. Defects Discovered for JTidy
Figure 9.11: Mean Executions Reviewed vs. Defects Discovered for JTidy $k = \{0, 1, 3\}$

JTidy, shown in Figure 9.10, showed a greater change between RAF and RRCS. RAF and RRCS perform similarly for the first 4 (of 8) defects. To discover 4 defects, with uniform mislabeling and $p_{ml} = 0.15$, RRCS required 19.71 reviews, where RAF required 30.14 reviews. To discover 6 defects, RAF reviews more than double the executions that RRCS does (83.09 for RAF vs 39.05). To discover all 8 defects, RAF must review 56% of all FAILURE-labeled executions (801.4 of 1414.1 executions). By contrast, RRCS must review an average of 263.34 (only 18.6%). As $p_{mf}$ increases, the difference between RAF and RRCS increases. At $p_{mf} = 0.25$, RAF reviews 806.09 more executions than RRCS, an increase of 37.5%. As $p_{mf}$ increases, RRCS does significantly better than RAF.

These differences shrink under differential mislabeling. To discover all defects with $p_{mf} = 0.15$. RAF reviews 48% of FAILUREs. RRCS reviews only 23%. At $p_{mf} = 0.25$, these numbers are 53.7% and 19% for RAF and RRCS, respectively.
Figure 9.12: Mean Executions Reviewed vs. Defects Discovered for ROME

$k$-NN ranking, shown in Figure 9.11, improves on RRCS. Under uniform mislabeling with $p_{ml} = 0.25$, $k$-NN ranking with $k = 1$ reviews more executions than RRCS (391.37 compared to 325.18 for 8 defects). With $k = 3$, however, RRCS discovers all 8 defects with 309.59 executions reviewed. With differential mislabeling ($p_{mf} = 0.25$), RRCS reviews 233.49, 233.43, and 199.65 executions to discover 8 defects without $k$-NN ranking, with $k = 1$, and with $k = 3$.

Figure 9.12 shows the RRCS/RAF results for ROME-F and ROME-L. ROME-F provides a modest improvement over RAF. Under uniform mislabeling with $p_{ml} = \{0.15, 0.20, 0.25\}$, RRCS with function-call profiling (ROME-F) reviews 34.13, 55.61, and 14.95 fewer executions than RAF to discover all 7 defects. This is a 1-5% improvement over RAF. When using line-coverage profiling (ROME-L 5%), on the other hand, RRCS reviews 198.77, 200.11, and 301.56 fewer executions than RAF under the same conditions; an improvement of between 17-22%. Simply changing the profiling...
technique has a noticeable impact on RRCS.

As with GCC, Javac, and JTidy, the relationship between RAF, RRCS with ROME-F, and RRCS with ROME-L did not change under differential mislabeling. Instead, each technique reviews fewer executions. With ROME-F, RRCS finds all 7 defects while reviewing 21.27, 23.73, and 39.87 fewer executions than RAF when \( p_{mf} = \{0.15, 0.20, 0.25\} \); a 3% to 5% difference. Using line coverage (ROME-L 5%), RRCS reviews 21.2%, 24.4%, and 18.3% fewer executions than RAF (a difference of 120.52, 164.39, and 142.77 executions with the same \( p_{mf} \) values).

Figure 9.13 shows the mean executions reviewed vs. defect discovered for RRCS with \( k \)-NN ranking for both ROME-F and ROME-L. Under either mislabeling model with ROME-F, \( k \)-NN ranking provides little benefit. When using ROME-L, \( k = 1 \) reviews 35.92, 86.01, and 45.42 fewer failures than plain RRCS under uniform mislabeling with \( p_{mf} = \{0.15, 0.20, 0.25\} \). This is an improvement of between 3 and
7.5%. Under differential mislabeling, RRCS with 1-NN ranking saved 6.5% (a savings of 51.14) over plain RRCS.

RRCS performs consistently well on Xerces over all mislabeling rates and mislabeling models. As shown in Figure 9.14, RRCS reviews between 29.1% and 32.9% fewer executions than RAF for $p_{ml} = \{0.15, 0.20, 0.25\}$ with uniform mislabeling. Under differential mislabeling, RRCS reviews between 23% and 24.7% fewer executions than RAF.

$k$-NN ranking has a small effect on RRCS. RRCS with $k = 1$ reviews .91 and .53 fewer executions than plain RRCS with uniform mislabeling and $p_{mf} = 0.15$ and 0.20, respectively. With $p_{ml} = 0.25$, 3-NN ranking reviews 8.45 fewer executions than plain RRCS (with lower $p_{ml}$, $k = 3$ reviews more executions than $k = 0$).
Figure 9.15: Mean Executions Reviewed vs. Defects Discovered for Xerces 5% $k = \{0, 1, 3\}$
Chapter 10

Conclusions and Future Work

This dissertation presents three novel techniques which developers can use to prioritize user-submitted feedback and execution profiles for review. The techniques are designed to handle two main usage scenarios. In the first scenario, the development organization is risk-averse: they have reason to believe that the defects present in their application will result in harm to their organization (in terms of future sales, reputation, etc) and to their clients (due to lost or corrupted data, or some catastrophic operation). In this scenario, the organization has both the resources and need to review executions as long as their is a justifiable reason to do so. In the second scenario, an organization either does not have the resources to explore a great many executions, or has reason to believe that any defects present are not serious. In this second scenario, developers are reviewing user-reported problems but wish to discover more failures than through random selection.

Developers need manageable mislabeling. The experimental results show that CBF, RAF+$k$NN, and RRCS are all relatively robust in the face of high mislabeling. Increased mislabeling requires developers to review more executions in order to discover the same number of failures and defects. A high rate of mislabeling may also indicate that the application’s functionality is confusing or overly complex (this may
itself be an application defect).

These techniques depend on several factors. Developers will need to collect useful execution profiles. Depending on how the application is used and the nature of the defects in it, one type of profiling may provide superior results. This was illustrated with the ROME dataset, where the use of function-call profiling caused all the methods to perform, at best, similarly to RAF. Using line-coverage profiling, on the other hand, the results were greatly improved. With ROME, the input data tends to be very similar: ROME parses and generates RSS and Atom feeds, which tend to collect the same types of data even though the data came from disparate sources. This means that most of the input data causes the same set of functions to execute. Selecting an appropriate type of profiling should be guided by knowledge of the application domain and the historical performance of the development team. In the first case, certain problem domains may point to specific types of profiling — for instance, developers may want to use data or information flow profiling to discover certain classes of security vulnerabilities. In the second, knowing that developers are prone to particular types of defects may point to the use of one form of profiling over another. If possible, development organizations may want to use Capture/Replay, which allows for online capture of program executions that can then be played back offline. During the offline replay, different profiling techniques can be used to determine the most efficacious one.

Finally, the different techniques may require some tuning for the underlying algorithms, such as the number of clusters for $k$-means and the number of neighbors for $k$ nearest neighbors. In practice, selecting reasonable values worked well (10\%, 20\%, and 30\% of the executions for the number of clusters, and 1, \ldots, 5 for the number of neighbors). Since developers can only verify these values by reviewing users’ labels, the best course of action is to pick a value that represents how much work the organization can afford to spend reviewing executions. Developers can then use more
advanced techniques to adjust this value if too many false positives are produced.

The first presented technique, Corroboration-Based Filtering (CBF) addresses the first risk-averse scenario. Developers begin by reviewing reported FAILUREs, but then expand their search to SUCCESS-labeled executions that are “suspicious”. For CBF, a SUCCESS is suspicious if it is either in the same cluster as a confirmed failure, or if it is in a very small cluster. Small clusters are defined by a corroboration threshold, T. In repeated experiments, CBF consistently discovered more failures and defects than the naive Review-All-FAILUREs method (RAF). In terms of the number of reviews required to reveal a failure (R/F<sub>d</sub>), however, CBF tended to be more expensive than RAF, especially at lower mislabeling rates. Higher threshold values (T > 1), in particular, caused CBF to be considerably more expensive than RAF. For all mislabeling rates, the lowest R/F<sub>d</sub> values for CBF were achieved by setting T = 0. It should be noted, however, that R/F<sub>d</sub> is not the only useful metric for comparing CBF and RAF. CBF with non-zero threshold values allow developers to discover defects contained in very small clusters. If these defects are very expensive, then discovering them may be worth the extra work required.

The second technique, Review-All-FAILUREs and k-Nearest Neighbors, also addresses the first scenario. The initial study of CBF (Chapter 6) showed that while CBF finds more failures and defects than the naive RAF strategy, it does so at increased cost in executions reviewed. The hope was that RAF+kNN would find nearly as many failures and defects as CBF, but do so at reduced cost (in terms of executions reviewed). In practice, RAF+kNN tended to be useful under limited circumstances. In terms of the number of executions reviewed required to discover a failure, RAF+1NN and RAF+3NN were more efficient than CBF when the mislabeling was below 20%. With mislabeling greater than 20%, CBF was often more efficient than all of the RAF+kNN variants. RAF was more efficient than both CBF and RAF+kNN when mislabeling was 10% or less. There was little reason to use
RAF+5NN: it did not find more failures or defects compared with CBF, but required as many executions reviewed. RAF+\(k\)NN did not discover as many defects as CBF with \(T = 1\), especially as mislabeling increased. CBF is the better choice if developers need to discover as many defects as possible.

While studying RAF+\(k\)NN (Chapter 7), we also introduced different mislabeling models. The original mislabeling model, dubbed *uniform mislabeling*, assumed that successful and failing executions were equally likely to be mislabeled. In Chapter 7, we introduced *differential mislabeling*. With differential mislabeling, failures and successes are mislabeled at different rates: while comparing CBF, RAF+\(k\)NN, and RAF, we mislabeled failures as in Chapter 6, but successes were mislabeled at a constant rate of 10\%. Differential mislabeling assumes that successful executions are easier for users to understand, and that they will then make fewer mistakes when mislabeling those executions. Differential mislabeling did not change number of failures and defects discovered by any of the methods. Instead, it caused the number of executions reviewed to *decrease* as \(p_{mf}\) (the probability that failures are mislabeled) increased. This had the effect of making RAF+\(k\)NN with \(k = \{1, 3\}\) more efficient than both CBF and RAF for moderate to high mislabeling (\(p_{mf} > 0.10\)).

In the studies of CBF and RAF+\(k\)NN, CBF with \(T = 1\) is recommended when developers wish to discover as many defects as possible. The prime use case for this is that the application contains costly defects that make the extra work required by higher threshold values worthwhile. In Chapter 8, we explore the *relative costs* of selecting a particular method (CBF with \(T = 1\), RAF+\(k\)NN with \(k = \{1, 3, 5\}\), and RAF), measured in terms of the amount of work required (in terms of executions reviewed) and the number of defects missed. The rationale in this study was that, while a technique such as RAF may require the least work in terms of developer time, it incurs a cost caused by undiscovered defects. If those undiscovered defects cost more than the amount of work expended, then selecting a more expensive (needs
more developer time) method that discovers more defects may be warranted. This total cost is presented as the sum of the *review cost units* required by a method and the number of missed defects multiplied by a multiplier \(M\). \(M\) is the number of review cost units each missed defect is worth. CBF and RAF+\(k\)NN tended to be the least costly methods when both mislabeling and \(M\) was relatively high.

The third presented technique, *round-robin cluster sampling with \(k\)-\(NN\) ranking* (RRCS), is designed for the second scenario: developers are only willing to review reported failures, and wish to minimize the number of successful executions they review. RRCS reviews the same set of executions as RAF (executions labeled FAILURE), but does so in a systematic manner: FAILUREs are grouped into clusters; one execution is selected from each cluster for review, and this process is repeated until the FAILUREs are exhausted. If \(k\)-\(NN\) ranking is used, then, within each cluster, executions are ranked for review based on their proximity to confirmed failure. At all mislabeling rates, RRCS discovered both failures and defects more quickly than RAF. In any situation requiring RAF, RRCS is a more efficient replacement.

All three techniques allow developers to take advantage of user-submitted feedback and profile data. By using user labels in conjunction with the groupings provided by clustering and nearest-neighbor techniques, developers can discover more failure and defects than Review-All-FAILUREs (which relies solely on user labeling) and techniques like One-Per-Cluster sampling (which only uses submitted profiles). The presented studies illustrate three points:

1. Both CBF and RAF+\(k\)NN discover considerably more failures than RAF. Either technique is suitable if developers need numerous examples of a set of defects.

2. CBF with \(T = 1\) find more defects than both RAF+\(k\)NN and RAF. CBF is especially useful when defects are costly: the extra work required by developers is more than offset by the cost by discovering additional defects.
3. When defects are not so costly as to require either CBF or RAF+$k$NN, RRCS is a suitable replacement for RAF: it reviews the same set of executions, but does so in a systematic fashion that discovers failures and defects more quickly.

### 10.1 Future Work

In the future, we intend to conduct additional empirical studies with executions actually submitted and labeled by users. These studies would examine both what the mislabeling rates for successes and failures are, and whether or not different regions of the profile space have different mislabeling rates.

In addition, it would be interesting to solicit *structured feedback* from users. This feedback would take the form of questions answerable by drop-down box or check box. By collecting information describing users' failures, developers gain an important debugging tool that may aid in fault localization and help to determine appropriate profiling techniques for use with capture/replay.

It may be possible to improve on the unsupervised techniques presented here. In particular, supervised learning techniques should help to filter out mislabeled executions. It will be easier to develop supervised learning techniques once we have examples of real user labeling.

These techniques should be integrated into both a software application and the issue tracking system used by the developers. By adding these techniques and data collection functionality to a popular open source bug tracking system, it would be easier to conduct future experiments. Conducting future experiments could be made easier by integrating these techniques and data collection functionality into a popular open source bug tracking system.
Bibliography


[5] Nicholas L. Crookston and Andrew O. Finley. yaimpute: An r package for knn


analysis of execution profiles. In ICSE ’01: Proceedings of the 23rd International
Conference on Software Engineering, pages 339–348, Washington, DC, USA,

distribution of program failures in a profile space. In ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th
ACM SIGSOFT international symposium on Foundations of software engineering,
http://dx.doi.org/10.1145/503209.503243. URL http://dx.doi.org/10.1145/
503209.503243.


classifying software failures. In Software Reliability Engineering, 2004. ISSRE


