UC Santa Cruz
UC Santa Cruz Electronic Theses and Dissertations

Title
Efficient Bug Prediction and Fix Suggestions

Permalink
https://escholarship.org/uc/item/47x1t79s

Author
Shivaji, Shivkumar

Publication Date
2013

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

EFFICIENT BUG PREDICTION AND FIX SUGGESTIONS

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Shivkumar Shivaji

March 2013

The Dissertation of Shivkumar Shivaji
is approved:

Professor Jim Whitehead, Chair

Professor Jose Renau

Professor Cormac Flanagan

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Copyright © by

Shivkumar Shivaji

2013

Table of Contents

List of Figures vi

List of Tables vii

Abstract viii

Acknowledgments x

Dedication xi

1 Introduction 1

1.1 Motivation . . . 1
1.2 Bug Prediction Workflow . . . 9
1.3 Bug Prognosticator . . . 10
1.4 Fix Suggester . . . 11
1.5 Human Feedback . . . 11
1.6 Contributions and Research Questions . . . 12

2 Related Work 15

2.1 Defect Prediction . . . 15
2.1.1 Totally Ordered Program Units . . . 16
2.1.2 Partially Ordered Program Units . . . 18
2.1.3 Prediction on a Given Software Unit . . . 19
2.2 Predictions of Bug Introducing Activities . . . 20
2.3 Predictions of Bug Characteristics . . . 21
2.4 Feature Selection . . . 24
2.5 Fix Suggestion . . . 25
2.5.1 Static Analysis Techniques . . . 25
2.5.2 Fix Content Prediction without Static Analysis . . . 26
2.6 Human Feedback . . . 28

3 Change Classification **30**
3.1 Workflow . . . 30
3.2 Finding Buggy and Clean Changes . . . 32
3.3 Feature Extraction . . . 34

4 Performance Metrics **37**

5 Bug Prognosticator **43**
5.1 Introduction . . . 43
5.2 Research Questions . . . 47
5.3 Feature Selection . . . 49
5.4 Feature Selection Techniques . . . 50
5.5 Feature Selection Process . . . 56
5.6 Experimental Context . . . 57
5.7 Results . . . 58
5.7.1 Classifier performance comparison . . . 58
5.7.2 Effect of feature selection . . . 63
5.7.3 Statistical Analysis of Results . . . 66
5.7.4 Feature Sensitivity . . . 68
5.7.5 Breakdown of Top 100 Features . . . 72
5.7.6 Alternative to 10-fold Validation . . . 74
5.7.7 Alternatives to using the Standard Buggy F-measure . . . 75
5.7.8 Algorithm Runtime Analysis . . . 76
5.8 Comparison to Related Work . . . 78
5.8.1 Feature Selection . . . 81
5.9 Threats to Validity . . . 82
5.10 Conclusion . . . 84

6 Bug Fix Suggester **95**
6.1 Introduction . . . 95
6.2 Research Questions . . . 98
6.3 Corpus . . . 99
6.4 Methodology . . . 99
6.4.1 Updates to Change Classification . . . 100
6.4.2 Fix Suggester . . . 103
6.5 Results . . . 108
6.5.1 Fix Suggester . . . 108
6.5.2 Breakdown of Top 100 Bug Inducing Features . . . 110
6.5.3 Algorithm Time Analysis . . . 111
6.6 Comparison to Related Work . . . 112
6.6.1 Static Analysis Techniques . . . 112
6.6.2 Fix Content Prediction without Static Analysis . . . 113
6.7 Threats to Validity . . . 114

6.8 Conclusion 115

7 Human Feedback **119**
7.1 Introduction 119
7.2 Human Feedback on Fix Prediction 120
7.2.1 Sample Questions Posed during the Human Feedback Process . . 125
7.3 Fix Suggester User Study 128
7.3.1 Research Questions 129
7.3.2 User Study Procedure 129
7.3.3 User Study Questions 131
7.4 Results 140
7.4.1 Human Feedback Improvement on Fix Prediction 140
7.4.2 Fix Suggester Qualitative Study 144
7.5 Threats to Validity 148

8 Future Work **150**
8.1 Introduction 150
8.2 Opportunities 150
8.3 Human Feedback 151
8.4 Fix Suggester 152
8.5 Bug Prognosticator 152
8.6 General 153

9 Conclusions **155**

Bibliography **157**

List of Figures

1.1 Developer Interaction Workflow 9
1.2 Enhanced Change Classification with the Bug Prognosticator 10
1.3 Fix Suggester 11

5.1 Classifier F-measure by Project 69
5.2 Buggy F-measure versus Features using Naïve Bayes 71

6.1 Developer Interaction Workflow 100
6.2 Fix Suggester 105
6.3 Fix Suggester F-measure on Project History 109

7.1 Example SVM Classifier in 2D 121
7.2 RQ8 Feedback 147
7.3 RQ9 Feedback 148

List of Tables

1.1 Summary of Methods To Manage Bugs 6

3.1 Example bug fix source code change 33

5.1 Feature groups. Feature group description, extraction method, and example features. 51
5.2 Summary of Projects Surveyed 52
5.3 Average Classifier Performance on Corpus (ordered by descending Buggy F-measure) 60
5.4 Naïve Bayes (with Significance Attribute Evaluation) on the optimized feature set (binary) 88
5.5 SVM (with Gain Ratio) on the optimized feature set (binary) 89
5.6 Naïve Bayes with 1% of all Features 90
5.7 Top 100 Bug Predicting Features 91
5.8 Temporal Validation of the data sets of Table 5.4 92
5.9 Alternative F-measures on the data sets of Table 5.4 93
5.10 Comparisons of Our Approach to Related Work 94

6.1 Example Bug Fix 116
6.2 Projects 117
6.3 Bug Fix Matrix Example 117
6.4 Average Fix Content Prediction rate per project from Project History 117
6.5 Top 100 Bug Reasons 118
6.6 Meta-data and Changelog Features in the Top 100 118

7.1 Average Bug Prediction Results after Feedback on 10 revisions per project 141

Abstract

Efficient Bug Prediction and Fix Suggestions

by

Shivkumar Shivaji

Bugs are a well known Achilles' heel of software development. In the last few years, machine learning techniques to combat software bugs have become popular. However, results of these techniques are not good enough for practical adoption. In addition, most techniques do not provide reasons for why a code change is a bug.

Furthermore, suggestions to fix the bug would be greatly beneficial. An added bonus would be engaging humans to improve the bug and fix prediction process.

In this dissertation, a step-by-step procedure which effectively predicts buggy code changes (Bug Prognosticator), produces bug fix suggestions (Fix Suggester), and utilizes human feedback is presented. Each of these steps can be used independently, but combining them allows more effective management of bugs. These techniques are tested on many open source and a large commercial project. Human feedback was used to understand and improve the performance of the techniques. Feedback was primarily gathered from industry participants in order to assess practical suitability.

The Bug Prognosticator explores feature selection techniques and classifiers to improve results of code change bug prediction. The optimized Bug Prognosticator is able to achieve an average 97% precision and 70% recall when evaluated on eleven projects, ten open source and one commercial.

The Fix Suggester uses the Bug Prognosticator and statistical analysis of keyword term frequencies to suggest unordered fix keywords to a code change predicted to be buggy. The suggestions are validated against actual bug fixes to confirm their utility. The Fix Suggester is able to achieve 46.9% precision and 38.9% recall on its predicted fix tokens. This is a reasonable start to the difficult problem of predicting the contents of a bug fix.

To improve the efficiency of the Bug Prognosticator and the Fix Suggester, active learning is employed on willing human participants. Developers aid the Bug Prognosticator and the Fix Suggester on code changes that machines find hard to evaluate. The developer's feedback is used to enhance the performance of the Bug Prognosticator and the Fix Suggester. In addition, a user study is performed to gauge the utility of the Fix Suggester.

The dissertation concludes with a discussion of future work and challenges faced by the techniques. Given the success of statistical defect prediction techniques, more industrial exposure would benefit researchers and software practitioners.

Acknowledgments

I want to thank my committee,

my research colleagues,

and many who were kind enough to give me feedback.

To my family.

Chapter 1

Introduction

1.1 Motivation

The global software market has an estimated market size of 267 billion dollars at the end of 2011. This is projected to reach 358 billion dollars in 2015 [127]. In this rapidly growing sector, software systems are produced and maintained by humans. Due to the fallibility of humans and the complexity of maintaining software, defects invariably creep into these systems. Most software companies spend much money and effort uncovering several bugs in a software system before releasing it to customers.

It is well known that software bugs constitute a huge burden for software development firms. Software verification is a tough process [163]. To better manage defects created by humans, additional personnel are hired as software testers. Kaner et al. report that many of the firms they surveyed have a 1-1 developer to tester ratio [36]. These include large firms such as Microsoft.

It is clear that software development firms are strongly investing in software quality if they are willing to hire as many testers as engineers. Software development costs are a significant share of the almost 300 billion dollar global software market. A huge portion of that amount is spent on testing software.

Despite spending large amounts on testing software, many defects are still present in the final product and can have a devastating consequence on a firm's reputation. Telang et al. show that a single production security bug can reduce a large company's stock value by 0.6% and can have a more powerful impact on smaller firms [152]. To reduce damage to a firm, it is important that software bugs are addressed as early as possible.

A rough outline of the typical stages in a software development process modeled after the famous but strict Waterfall model [137] is given below:

1. Gathering Requirements
2. Design
3. Implementation
4. Integration
5. Verification
6. Product Release
7. Post-Release Maintenance

In the traditional waterfall model, each stage is performed in isolation, often lasting months. After the end of a stage, a quality check was performed before moving on to the next one. Modern software projects might operate more dynamically and prefer more flexible development processes. Newer software development processes have more interplay amongst the stages, and the duration of a particular stage might be as short as a few days. However, the stages listed above are still present in any process, as all software needs to be designed, implemented, integrated, verified, and released.

A defect caught after a product release is expensive as the damage can be seen by a customer and likely the general public. Defects caught close to the release date are still expensive. Developers are typically scrambling at this stage to ensure that deadlines are being met. A presence of a bug at this stage is highly likely to impact the final release date. Tracing backwards in the list, it becomes clear that the earlier a defect is addressed, the more collective employee time and money are saved.

A common belief in industrial software engineering is that software defects cost logarithmically more to fix, the later they are found in the software development life cycle. Boehm et al. stated in 1988, that a defect appearing in production can cost 50-200 times as much to fix than if it was corrected earlier [32]. In 2001, Boehm et al. revised this ratio to 5:1 for small non-critical systems, instead of the earlier 100:1 [31]. As we cannot expect all systems to be small and non-critical, practical defect costs are noticeably higher the longer they are ignored.

The research presented in this dissertation focuses on addressing bugs at a reasonably early stage, the implementation stage. Table 1.1 contains detailed information

on the efficacy of popular techniques to fight bugs during this stage.

Unit Tests These are tests to ensure that an individual unit of code performs satisfactorily independent of other code modules.

Unit tests are a good way to fight easy bugs but have a large setup time as developers need to program and configure a unit test from scratch. In addition, maintenance costs on unit tests are expensive as they need to be constantly updated to continue to be accurate tests. New features/interactions can potentially break existing unit tests. However, unit tests are typically quick to run.

System Tests These are more comprehensive end to end tests that verify full system functionality.

They take a while to setup, a while to execute, and also are expensive to maintain. However, they are better predictors of hard bugs compared to unit tests.

Code Inspections Before submitting complex code to a source repository, an engineer often asks a coworker to perform a code review. The idea is bugs within a code change will be less likely to escape two minds.

Code inspections can provide reasonable utility on hard bugs but are expensive as they make use of people's time. This is compounded by the fact that there might be only a few qualified people capable of conducting reviews on a particular code change.

Static Analysis Tools can analyze code grammar and advise programmers of errors

in logic.

Static analysis can quickly point out obvious flaws in logic. It also has low setup and execution times. However, a lot of false positives are produced. Kim et al in [87] show that less than 5% of bugs revealed by static analysis are fixed. Thus, static analysis typically finds bugs that are not critical.

Dynamic Analysis While running the program, certain types of errors can be caught. Examples include memory leaks and code performance issues.

Dynamic analysis can help find hard bugs, but requires a considerable amount of human interaction, and a huge setup cost as it is important to know what we are looking for before starting a dynamic analysis session. Additionally, dynamic analysis and system tests do not precisely point out the location of the faulty code.

All of the above techniques represent a way to reduce future bugs. However, they suffer from multiple limitations and do not indicate faults at a low level of granularity, namely at the method or line of code level. An exception is code inspections which are expensive due to needing humans.

In contrast, the Bug Prognosticator presented in chapter 5, is able to present the location of faulty code, with a low execution cost, and can be used on hard and easy bugs. In addition, the prediction of faulty code is made at the time of the commit, thus alerting developers as early as possible. The setup cost is somewhat expensive for large projects, however it is a one time cost for a software project.

The Bug Prognosticator is based on statistical bug prediction techniques. Sta-

Technique	Setup Time	Execution Time	Utility on Easy Bugs	Utility on Hard Bugs	Utility in Predicting Defect Location	Maintenance Cost
Unit Tests	High	Low	Good	Poor	Good	High
System Tests	High	High	Poor	Fair	Poor	High
Code Inspection	High	Medium	Good	Fair	Good	Low
Static Analysis	Low	Low	Fair	Poor	Good	Low
Dynamic Analysis	High	High	Poor	Fair	Poor	Low

Table 1.1: Summary of Methods To Manage Bugs

tistical bug prediction operates by looking for patterns in code that correlate with (or rather predict) bugs. Typically, statistical techniques do not extensively utilize program introspection techniques. Text from code or abstract syntax trees are often treated as a bag of words. The general benefits of a statistical approach are:

- A focus on predicting bugs that will actually be fixed in practice. Wedyan et al. have empirically analyzed bug reports by static analysis tools and found that less than 3% of the suggestions are actually fixed in practice [156]. When interacting with a few industrial settings, this number was found to be less than 0.5%.

- Project history is often leveraged and tailored for adaptive prediction of bug fixes relevant to the future code changes of a particular project. Historical trends can also be exploited. If a particular type of bug fix was popular at the onset of a project but diminished in significance soon, statistical fix content prediction will downplay the importance of that fix pattern.

The benefits to program analysis when compared to statistical approaches including:

- Diagnosis typically reflects a clear defect that can be triggered on a program execution path. In contrast, statistical approaches predict a program unit is buggy probabilistically. As statistical approaches work probabilistically, they can only predict the likelihood of a defect.

- Human understandable reasons for a bug prediction are typically provided.

Bug Prognosticator is an optimized statistical bug predictor on a code change. To further speed up the bug fix process, which on average takes fourteen hours per fix after a release [146], our solution helps understand the bug and provides assistance to fix it.

In order to predict bugs, a mechanism that accurately classifies a code change and indicates the top reasons for a code change being buggy or clean is of much value. Chapter 5 introduces Bug Prognosticator, which classifies a code change as buggy or clean using statistical methods. These improvements greatly enhance the speed and accuracy of this technique.

If a code change is found to be buggy, top reasons for the bug are extracted by the Bug Prognosticator. These are passed to the Fix Suggester in chapter 6, which then computes programming language tokens that are most likely to appear in a bug fix to the given buggy code change. Finally, these bug fix suggestions will be investigated by human experts to certify their validity. Their feedback is returned to the Bug Prognosticator and Fix Suggester. The performance of the human optimized Bug Prognosticator and Fix Suggester is compared to the pure machine driven approach of chapter 5 in chapter 7. The next section addresses relevant research questions for the approaches presented in the dissertation.

The next section illustrates the overall workflow with the Bug Prognosticator, Fix Suggester, and Human Feedback working in concert. Each component is briefly distilled to give the reader a background before contributions and research questions are outlined.

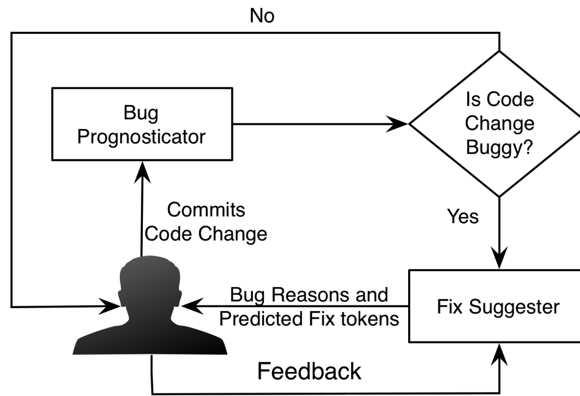


Figure 1.1: Developer Interaction Workflow

1.2 Bug Prediction Workflow

An overview of the developer interaction workflow is depicted in Figure 1.1. The steps of the process are:

- A code change is submitted.
- A prediction is made on whether the entire change is buggy or clean using the Bug Prognosticator. The Bug Prognosticator performs an optimized code change classification on change history [87, 144] to predict if a code change is buggy or clean. A summary of the optimized change classification is presented in section 1.3. Change classification is detailed in chapter 3.
- If the code change is predicted to be buggy, suggest a partial code fix and expose reasons for the bug to the developer. The predicted fix tokens are also presented to the user. The Fix Suggester is briefly described in Section 1.3.

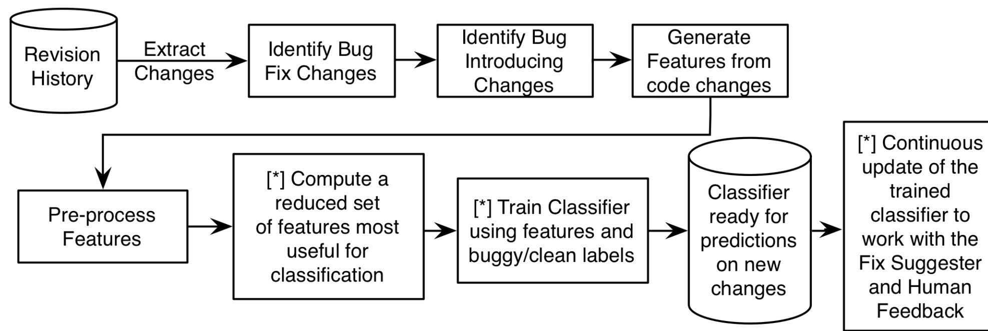


Figure 1.2: Enhanced Change Classification with the Bug Prognosticator

- The user can provide feedback to the Bug Prognosticator and the Fix Suggester. This will help refine both the current and future predictions.

1.3 Bug Prognosticator

The change classification process is distilled in figure 1.2. The steps marked with an asterisk are enhancements to the change classification process introduced by the Bug Prognosticator. Change classification is fully described in chapter 3.

At a high level, past revision history is used to gather prior buggy and clean changes. Features are extracted from these changes. A classifier is trained with the code changes and labels. Future code changes can then be predicted as buggy or clean.

If a code change is predicted as buggy, reasons for that decision are used to compose a partial bug fix. The next section briefly explores how this is done by the Fix Suggester.

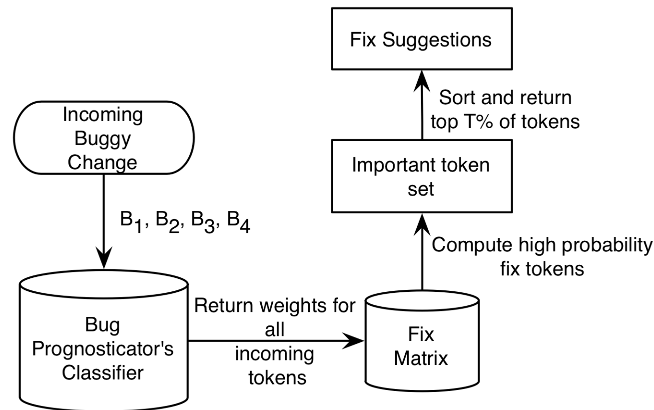


Figure 1.3: Fix Suggester

1.4 Fix Suggester

Figure 1.3 graphically introduces the Fix Suggester. Once a code change is predicted as buggy, the Bug Prognosticator will provide reasons for the prediction. The reasons are sent as token weights for the incoming code change to the Fix Suggester. Internally, a Fix matrix is built using the token weights and past change data. The Fix matrix then computes terms which are likely to occur in the bug fix to the incoming code change. These terms are ranked and filtered before passing them on as fix suggestions to a developer.

1.5 Human Feedback

Developers have an opportunity to contest bug reasons and the proposed fix suggestions. Providing the optimal feedback mechanism is a tough challenge in practice.

Chapter 7 employs active learning to correct initial labeling of a code change as buggy or clean. Interestingly, using human feedback on a few revisions was able to significantly improve performance of both the Fix Suggester and the Bug Prognosticator. Chapter 7 also conducts a user study on the practical utility of the Fix Suggester. The fix suggestions were generally assessed by users to be useful.

The next section addresses the specific contributions and research questions.

1.6 Contributions and Research Questions

1. The Bug Prognosticator - an optimized method based on feature selection to accurately predict if an impending code change contains a bug. The Bug Prognosticator also updates the classifier to produce better input to the Fix Suggester.

   - *RQ1. Which variables lead to best bug prediction performance when using feature selection?*

     The three variables affecting bug prediction performance that are explored in this dissertation are: (1) type of classifier (Naïve Bayes, Support Vector Machine), (2) type of feature selection used (3) and whether multiple instances of a feature are significant (count), or whether only the existence of a feature is significant (binary).

   - *RQ2. Range of bug prediction performance using feature selection.* How do the best-performing classifiers perform across all projects when using feature selection?

- *RQ3. Feature Sensitivity.* What is the performance of change classification at varying percentages of features? What is the F-measure of the best performing classifier when using just 1% of all project features?
- *RQ4. Best Performing Features.* Which classes of features are the most useful for performing bug predictions?

RQ1-4 are addressed in chapter 5.

2. Fix Suggester - A method to predict partial content of a bug fix given a bug inducing code change and input from the Bug Prognosticator. Throughout the dissertation, the "content" of a bug fix refers to keywords of a bug fix.

- *RQ5. What is the prediction accuracy of the Fix Suggester on varying points of project history?*
- *RQ6. What kind of programming tokens are ideal for generating Fix Suggestions?*

RQ5-6 are addressed in chapter 6.

3. A method to further optimize the results of the previous contribution using human feedback.

- *RQ7. Using human feedback, how much can we improve on the results of Fix Suggester?*

4. A qualitative study to gauge the practical utility of the Fix Suggester.

- *RQ8. When engineers inspect the bug fix change log, do they find that the Fix Suggester's keywords are relevant to the actual bug fix?*
- *RQ9. Does reviewing the Fix Suggester's keywords influence the investigation for the bug fix?*

RQ7-9 are addressed in chapter 7.

The next chapter starts with a survey of related work. Following, the notion of bug prediction on a code change, change classification is introduced. Next, performance metrics are detailed. The stage is now set to discuss contributions. The first contribution of improved bug prediction on a code change, the Bug Prognosticator, is presented. The contribution of the Fix Suggester is then presented. An attempt to improve the Bug Prognosticator and Fix Suggester using human feedback is the next contribution. Finally, a qualitative study of the Fix Suggester is performed. Opportunities for future work are then discussed followed by the conclusion.

Chapter 2

Related Work

The major contributions of this dissertation address bug prediction, human feedback, and fix suggestion. This chapter, reviews related work in each of these areas while also touching upon broader areas. We start with a survey of defect prediction. Some of these techniques were enhanced to have better results. Related work on feature selection is discussed in order to better understand the results of chapter 5. Next, related work on human feedback is addressed. We start from a broader level before proceeding to related work within software engineering. Finally, related work on bug fix suggestions is investigated. We start with motivating examples from expert systems before moving to bug fixes on software artifacts.

2.1 Defect Prediction

Given a software project containing a set of program units (files, classes, methods or functions, or changes depending on prediction technique and language), a bug

prediction algorithm typically outputs one of the following.

Totally Ordered Program Units. A total ordering of program units from most to least bug prone [85] using an ordering metric such as predicted bug density for each file [126]. If desired, this can be used to create a partial ordering (see below).

Partially Ordered Program Units. A partial ordering of program units into bug prone categories (e.g. the top $N\%$ most bug-prone files in [88, 72, 126])

Prediction on a Given Software Unit. A prediction on whether a given software unit contains a bug. Prediction granularities range from an entire file or class [73, 69] to a single change (e.g., Change Classification [87]).

Bug prediction algorithms can also operate at a higher level to output:

Predictions of Bug Introducing Activities. Suspect behaviors or plans that lead to defect introduction.

Predictions of Bug Characteristics. Information that can help identify characteristics of a particular defect.

Related work in each of these areas is detailed below. As chapter 5 employs feature selection, related work on feature selection techniques is also addressed.

2.1.1 Totally Ordered Program Units

Khoshgoftaar and Allen have proposed a model to list modules according to software quality factors such as future fault density [86, 85]. The inputs to the model are software complexity metrics such as LOC, number of unique operators, and cyclomatic complexity. A stepwise multiregression is then performed to find weights for each factor.

Briand et al. use object oriented metrics to predict classes which are likely to contain bugs. They used PCA in combination with logistic regression to predict fault prone classes [34]. Morasca et al. use rough set theory and logistic regression to predict risky modules in commercial software [118]. Key inputs to their model include traditional metrics such as LOC, code block properties in addition to subjective metrics such as module knowledge. Mockus and Weiss predict risky modules in software by using a regression algorithm and change measures such as the number of systems touched, the number of modules touched, the number of lines of added code, and the number of modification requests [117]. Knab et al. apply a decision tree learner on a number of source code, modification, and defect measures to predict module defect density [92]. File defect densities were predicted with acceptable accuracy for the same release. They found that the LOC has little predictive power with regard to defect density. Instead, evolution data was a better predictor of defect density. Bernstein et al. propose a non-linear model on temporal features to predict the number and location of bugs in program code [23]. Using the Weka J48 decision tree learner, models were built for six plugins of the Eclipse project. The input to the machine learner included features such as the number of revisions and issues reported within the last few months. They conclude that a non-linear model in combination with a set of temporal features is able to predict the number and location of bugs with a high accuracy. Ostrand et al. identify the top 20 percent of problematic files in a project [126]. Using future fault predictors and a negative binomial linear regression model, they predict the fault density of each file. Data was gathered by linking VCS data and data from a modification request (MR)

database. As MRs may also contain information on new features or larger changes in addition to faults, they used a combination of heuristics and manual verification to improve data quality. Graves et al. analyze VCS history from a legacy telephone switching system [67]. They investigate how effective metrics from change history are for predicting fault distribution in software modules. They show that in general metrics based on change history are more useful in predicting fault rates when compared to traditional metrics. For example, the number of times a module has been changed is a better predictor of how many faults it will contain than its LOC or its McCabe complexity. They also found that older modules are less likely to contain faults than newer ones.

2.1.2 Partially Ordered Program Units

The previous section covered work which is based on total ordering of all program modules. This could be converted into a partially ordered program list, e.g. by presenting the top $N\%$ of modules, as performed by Ostrand et al. above. This section deals with work that can only return a partial ordering of bug prone modules. Askari and Holt investigate three probabilistic models that predict future modification of files based on available change histories of software [16]. This information was also used to predict which files will have bugs. A rigorous approach for evaluating predictive models derived from the Natural Language Processing domain is also provided. The accuracies of the three models are quite different.

Hassan and Holt use a caching algorithm to compute the set of fault-prone

modules, called the top-10 list [72]. They use four factors to determine this list: software units that were most frequently modified, most recently modified, most frequently fixed, and most recently fixed. A recent study by Shihab et al. [143] investigates the impact of code and process metrics on future defects when applied on the Eclipse project. The focus is on reducing metrics to a reach a much smaller though statistically significant set of metrics for computing potentially defective modules. Kim et al. proposed the bug cache algorithm to predict future faults based on previous fault localities [88]. In an empirical study of partially ordered faulty modules, Lessmann et al. [99] conclude that the choice of classifier may have a less profound impact on the prediction than previously thought, and recommend the ROC AUC as an accuracy indicator for comparative studies in software defect prediction.

2.1.3 Prediction on a Given Software Unit

Detailed comparison against relevant related work in this section is done in section 5.8 of chapter 5. They are briefly presented in this section for the sake of completeness.

Using decision trees and neural networks that employ object-oriented metrics as features, Gyimothy et al. [69] predict fault classes of the Mozilla project across several releases.

Aversano et al. [18] use KNN (K nearest neighbors) to locate faulty modules. Hata et al. [73] show that a technique used for spam filtering of emails can be successfully used on software modules to classify software as buggy or clean. Menzies et al. [113]

achieve good results on their best projects.

Kim et al. show that using support vector machines on software revision history information can provide good prediction accuracy. Elish and Elish [53] also used SVMs to predict buggy modules in software.

Recently, D'Ambros et al. [46] provided an extensive comparison of various bug prediction algorithms that operate at the file level using ROC AUC to compare algorithms. The ROC AUC metric is described in chapter 4.

Challagulla et al. investigate machine learning algorithms to identify faulty real-time software modules [37]. They find that predicting the number of defects in a module is much harder than predicting whether a module is defective. They achieve best results using the Naïve Bayes classifier. They conclude that "size" and "complexity" metrics are not sufficient attributes for accurate prediction.

2.2 Predictions of Bug Introducing Activities

Zimmerman et al. apply data mining to software revision histories in order to derive rules which alert programmers of related changes [171]. It is based on the spirit of "Programmers who changed these functions also changed .." Given a set of existing changes, these rules can suggest and predict likely changes, display item coupling that can escape program analysis, prevent changes due to incomplete changes. Zimmerman et al. provide a tool suggesting further changes to be made and warns about missing changes using historical data. After little training from software history, the tool could

can correctly predict 26% of further files to be changed and 15% of precise functions or variables. This is particularly useful for identifying missing co-changes.

The impact of design decisions on software quality is addressed by Schröter et al in a study of 52 Eclipse plugins [141]. They found that software design as well as past failure history can be used to build models that accurately predict fault-prone components in new programs. The prediction is based on usage relationships between software components. These are typically decided on before the coding phase. They achieve good accuracy in predicting the top 5% of most failure-prone components.

2.3 Predictions of Bug Characteristics

Predicting Security Bugs

Neuhaus et al. introduce Vulture, a tool that predicts vulnerable components in large software systems [124]. Vulture leverages project VCS history and a project vulnerability database to locate components that had past vulnerabilities. It then passes the import structure of software components as input to a support vector machine. After some training, the support vector machine can predict if a component is vulnerable or not based on its imports. Neuhaus et al. applied Vulture to the Mozilla project and found that two-thirds of vulnerable components were correctly identified. In practice, correctly predicting a bug as security related will help both in assigning it to the right team/developer and might increase the bug's priority.

In a related effort, Gegick et al. combat the problem of mislabeled security bugs [63]. They claim that due to the lack of security domain knowledge, many security bugs are mislabeled as non-security defects. To address this issue, Gegick et al. apply text mining algorithms on the bug description field. After training on many bug reports, the algorithms can classify a bug report as related to security or not. This model was evaluated on a large Cisco software system with over ten million lines of source code. A high percentage (78%) of non-security related bugs taken from a sample of bug reports should have classified as security related. Cisco security engineers validated several of these predictions, and Gegick et al.'s approach was shown to have high precision, 98% on the sample run.

Zaman et al. empirically investigate security, performance, and other bug types from the Firefox project [167]. They analyze differences in time to fix, developer experience, and bug fix characteristics among security, performance and other bug types. Security bugs require more developers with more experience when compared to other bug types. However, they need less triage time and are fixed faster than other bug types. Performance bugs also require more developers with more experience to address them. The bug triage time is comparable to a typical bug. This kind of analysis reveals utility in correctly classifying a bug type as early as possible.

Automatically classifying bug reports as security related, or performance can greatly improve efficiency in engineering organizations.

Defect or an enhancement?

Antoniol et al. investigate whether the text of bug reports is enough to classify the requested change as a bug or enhancement [9]. Decision trees, logistic regression, and naive Bayes classifiers are used to perform the classification. On Mozilla, Eclipse, and JBoss, they report an accuracy between 77 and 82 percent. The results appear promising, however it is useful to investigate if a bug reporter's text is itself influenced by whether the issue is a defect or an enhancement. For example, if the word fix appears in the text, the reporter might likely mark the issue as a bug.

Predicting the severity of a bug

Lamkanfi et al. attempt to predict the severity of a bug by analyzing the text of a bug report using text mining algorithms [96]. They report precision and recall figures ranging between 65-75% for Mozilla and Eclipse and 70-85% for Gnome. Automated means of predicting bug severity will help managers decide on how soon a bug needs to be fixed. Here too, it is useful to investigate if developers have a preconceived notion of the severity of a defect based on the report text. For example, the terms crucial and important might strongly indicate a bug of high severity. Nevertheless, research in this direction can help guard against bug severity misclassification.

Predicting who should fix a bug

Anvik et al. use machine learning to figure out which developer to assign a bug

to [10]. They perform the assignment based on the text in the bug report. They achieve precision levels of 57% and 64% on the Eclipse and Firefox projects respectively. Further research on this front can be useful for large engineering organizations. Fast assignment of a bug to the right person will typically speed up resolution.

Predicting the lifetime of a bug

Weiss et al. present a mechanism that predicts the amount of man-hours a bug fix will take [161]. Using the bug report as input, they return the average fix time from a set of similar historical bug reports. On the jBoss project, the predicted fix times were quite close to actual times. To be useful in practice, the predicted average fix times have to continue to be accurate throughout the duration of a project. Further research in this direction can help confirm if an adaptive real world solution is feasible.

The next section moves on to work focusing on feature selection. Feature selection was extensively performed in chapter 5.

2.4 Feature Selection

Detailed comparison against relevant related work in this section is done in section 5.8.1 of chapter 5. In similar fashion to section 2.1.3, related work is briefly presented in this section for the sake of completeness.

Hall and Holmes [70] compare six different feature selection techniques when

using the Naïve Bayes and the C4.5 classifier [132]. Each dataset analyzed has about one hundred features.

Song et al. [150] propose a general defect prediction framework involving a data preprocessor, feature selection, and learning algorithms. They also note that small changes to data representation can have a major impact on the results of feature selection and defect prediction.

Gao et al. [62] apply several feature selection algorithms to predict defective software modules for a large legacy telecommunications software system. They note that removing 85% of software metric features does not adversely affect results, and in some cases improved results.

2.5 Fix Suggestion

Suggestions for Bug Fixes can come from different techniques. The most common is via static analysis. Related work not using static analysis is also discussed.

2.5.1 Static Analysis Techniques

Predicting bug fix content is a challenging problem especially when using statistical techniques. The static analysis community has spent considerable effort in exploiting language semantics to suggest fixes to bugs. Popular tools using static analysis for fix suggestions include Findbugs [19], PMD [138], BLAST [121], FxCop [155] amongst many others. There are also approaches from literature which do not yet have downloadable tools available.

Demsky et al. focused on data structure inconsistency [48, 47]. Their approach checks data structure consistency using formal specifications and inserts run-time monitoring code to avoid inconsistent states. This technique provides workarounds rather than actual patches since it does not modify source code directly.

Arcuri et al. introduced an automatic patch generation technique [12, 15, 14]. They used genetic programming. Their evaluation was limited to small programs such as bubble sort and triangle classification.

2.5.2 Fix Content Prediction without Static Analysis

Brun et al [35] use machine learning to detect program properties which are known to result from errors in code. They rank and classify these properties from user written code. They then warn the user if many such properties are present.

Nguyen et al. [125] recommend bug fixes for a class or method if a fix was applied to a code peer. If a relevant bug fix was applied to a similar class, it is then recommended. Code peers are defined as objects which work in a similar manner when using a graph based representation of object usages.

Kim et al.'s Bugmem provides fix suggestions using past revision history [90] . Bug and fix pairs are extracted from history. If an impending code change is similar to a previous bug, the prior buggy change and bug fix are displayed, and the developer is warned. This is a useful tool especially for developers who are new to a code base. They can be alerted of mistakes from project history. The technique's limitation is the sheer amount of possible completions for a bug fix.

Weimer et al. use genetic learning to suggest a fix based on mutations of the ASTs of the source hunks [160]. If there is a comprehensive set of unit tests, and a bug was introduced which broke a unit test, this technique can repair that unit test using genetic learning and mutating AST of the source hunks until all unit tests pass again. The limitation of this technique is the assumption that unit tests are comprehensive and the fact that the fix suggested may not be optimal or even a feasible one.

Holmes and Murphy proposed an approach to extract structural components from example code and use them to assist coding when developers are working on similar code [75].

The approach presented in this chapter is similar to BugMem and Holmes et al. in making suggestions for a code change. The difference is that the recommended tokens are continuously validated against actual bug fixes. This ensures that the fix recommendations are likely to appear in actual fixes.

In the field of information retrieval, matrices are often used to represent document similarity [136]. While fix content prediction is a different problem, representing bugs and fixes as problem spaces and relating these spaces using a matrix has a high level parallel. The matrix relating buggy changes to fixes modeled a linear relationship in this chapter. Future work can extend the concept to capture more complex relationships between buggy and fix changes.

2.6 Human Feedback

While leveraging human feedback has been prevalent in many domains [41, 38] it has not been used extensively for understanding software bugs.

Active learning was used to classify software behavior in [33]. Active learning operates under the following assumptions.

- There are many unlabeled data points. The number of unlabeled data points dominate the amount of labeled ones.
- It takes a considerable amount of time to manually label a data point (using human effort).

More specifically, active learning [5] is a semi supervised model which queries humans iteratively on small units of unlabeled data to formulate a classifier. The benefits are a small subset of data can be used to construct a model, thus saving time and human effort. In addition, the active learner requests experts to create labels on tough to classify data points but not on easier ones. This is a more efficient use of human time and also makes the task more interesting for humans. The downside to active learning is the technique learns from a small amount of data points. If these points are invalid or illogical, the accuracy of the active learner will drop significantly.

Bowring et al [33] mention that classifying software behavior is a process that naturally contains many unlabeled data points and a huge cost to label each of the points. They also indicate that active learning provided results comparable to supervised learning in their experiments.

Xiao et al [166] use active learning for modeling software behavior in commercial games. Their work was tested on Electronic Arts' FIFA soccer game. Using active learning, the program changed its strategy when humans found easy ways to score goals. The learned strategies are then displayed to the user in a comprehensible format. Xiao et al. conclude that actively learning helped achieved good scalability for complex game scenarios while providing useful visual models to humans. To a certain extent, our goals in chapters 6 and 7 are similar, to create scalable models which aid human understanding of bugs in code.

A recent paper by Lo et al. incorporates incremental human feedback to continuously refine clone anomaly reports [106]. They first present the top few anomaly reports from a list of reports. Users can either accept or reject each presented clone anomaly report. After feedback on a few reports, the classifier is essentially rebuilt and the rest of the reports are re-sorted. The feedback algorithm used is at a batch level in contrast to the active learning technique of Xiao et al. When batch feedback is performed, a classifier is rebuilt after feedback on a set of data points is delivered. In active learning mode, the classifier is rebuilt or re-adjusted after feedback is delivered on a single point.

Chapter 3

Change Classification

Kim et al [87] introduced the change classification technique to help identify bug inducing changes. The motivation as mentioned before is the desire to classify a code change quickly and precisely as buggy or clean. Once a developer gets feedback that certain recent code changes are buggy, they can re-review those changes and potentially fix them on the spot. A workflow of this process is detailed in section 3.1.

Kim et al achieved an accuracy of 78% and a buggy F-measure of 60% when tested on twelve open source projects [87]. The definition of these performance metrics is described in chapter 4.

3.1 Workflow

The primary steps involved in performing change classification on a single project are as follows:

Creating a Corpus:

1. File level change deltas are extracted from the revision history of a project, as stored in its SCM repository (described further in Section 3.2).

2. The bug fix changes for each file are identified by examining keywords in SCM change log messages (Section 3.2).

3. The bug-introducing and clean changes at the file level are identified by tracing backward in the revision history from bug fix changes (Section 3.2).

4. Features are extracted from all changes, both buggy and clean. Features include all terms in the complete source code, the lines modified in each change (delta), and change meta-data such as author and change time. Complexity metrics, if available, are computed at this step. Details on these feature extraction techniques are presented in Section 3.3. At the end of Step 4, a project-specific corpus has been created, a set of labeled changes with a set of features associated with each change.

Classification:

5. A classification model is trained.

6. Once a classifier has been trained, it is ready to use. New code changes can now be fed to the classifier, which determines whether a new change is more similar to a buggy change or a clean change. Classification is performed at a code change level using file level change deltas as input to the classifier.

3.2 Finding Buggy and Clean Changes

The process of determining buggy and clean changes begins by using the Kenyon infrastructure to extract change transactions from either a CVS or Subversion software configuration management repository [25]. In Subversion, such transactions are directly available. CVS, however, provides only versioning at the file level, and does not record which files were committed together. To recover transactions from CVS archives, we group the individual per-file changes using a sliding window approach [172]: two subsequent changes by the same author and with the same log message are part of one transaction if they are at most 200 seconds apart.

In order to find bug-introducing changes, bug fixes must first be identified by mining change log messages. We use two approaches: searching for keywords in log messages such as “Fixed”, “Bug” [116], or other keywords likely to appear in a bug fix, and searching for references to bug reports like “#42233”. This allows us to identify whether an entire code change transaction contains a bug fix. If it does, we then need to identify the specific file delta change that introduced the bug. For the systems studied in this chapter, we manually verified that the identified fix commits were, indeed, bug fixes. For JCP, all bug fixes were identified using a source code to bug tracking system hook. As a result, we did not have to rely on change log messages for JCP.

The bug-introducing change identification algorithm proposed by Śliwerski, Zimmermann, and Zeller (SZZ algorithm) [149] is used in this chapter. After identifying bug fixes, SZZ uses a diff tool to determine what changed in the bug-fixes. The diff tool

Table 3.1: Example bug fix source code change

1.23: Bug-introducing	1.42: Fix
...	...
15: `if (foo==null) {`	36: `if (foo!=null) {`
16: `foo.bar();`	37: `foo.bar();`
...	...

returns a list of regions that differ between the two files; each region is called a "hunk". It observes each hunk in the bug-fix and assumes that the deleted or modified source code in each hunk is the location of a bug. Finally, SZZ tracks down the origins of the deleted or modified source code in the hunks using the built-in annotation function of SCM (Source Code Management) systems. The annotation computes, for each line in the source code, the most recent revision where the line was changed, and the developer who made the change. These revisions are identified as bug-introducing changes. In the example in Table 3.1, revision 1.42 fixes a fault in line 36. This line was introduced in revision 1.23 (when it was line 15). Thus revision 1.23 contains a bug-introducing change. Specifically, revision 1.23 calls a method within foo. However, the if-block is entered only if foo is null.

3.3 Feature Extraction

To classify software changes using machine learning algorithms, a classification model must be trained using features of buggy and clean changes. In this section, we discuss techniques for extracting features from a software project's change history.

A file change involves two source code revisions (an old revision and a new revision) and a change delta that records the added code (added delta) and the deleted code (deleted delta) between the two revisions. A file change has associated meta-data, including the change log, author, and commit date. Every term in the source code, change delta, and change log texts is used as a feature. This means that every variable, method name, function name, keyword, comment word, and operator—that is, everything in the source code separated by whitespace or a semicolon—is used as a feature.

We gather eight features from change meta-data: author, commit hour (0, 1, 2, . . . , 23), commit day (Sunday, Monday, . . . , Saturday), cumulative change count, cumulative bug count, length of change log, changed LOC (added delta LOC + deleted delta LOC), and new revision source code LOC.

We compute a range of traditional complexity metrics of the source code by using the Understand C/C++ and Java tools [76]. As a result, we extract 61 complexity metrics for each file, including LOC, number of comment lines, cyclomatic complexity, and max nesting. Since we have two source code files involved in each change (old and new revision files), we can use complexity metric deltas as features. That is, we can

compute a complexity metric for each file and take the difference; this difference can be used as a feature.

Change log messages are similar to e-mail or news articles in that they are human readable texts. To extract features from change log messages, we use the bag-of-words (BOW) approach, which converts a stream of characters (the text) into a BOW (index terms) [142].

We use all terms in the source code as features, including operators, numbers, keywords, and comments. To generate features from source code, we use a modified version of BOW, called BOW+, that extracts operators in addition to all terms extracted by BOW, since we believe operators such as !=, ++, and && are important terms in source code. We perform BOW+ extraction on added delta, deleted delta, and new revision source code. This means that every variable, method name, function name, programming language keyword, comment word, and operator in the source code separated by whitespace or a semicolon is used as a feature.

We also convert the directory and filename into features since they encode both module information and some behavioral semantics of the source code. For example, the file (from the Columba project) "ReceiveOptionPanel.java" in the directory "src/mail/core/org/columba/mail/gui/config/ account/" reveals that the file receives some options using a panel interface and the directory name shows that the source code is related to "account," "configure," and "graphical user interface." Directory and filenames often use camel case, concatenating words breaks with capitals. For example, "ReceiveOptionPanel.java" combines "receive," "option," and "panel." To extract such

words correctly, we use a case change in a directory or a filename as a word separator. We call this method BOW++.

The next chapter details performance metrics used in this dissertation before proceeding to the Bug Prognosticator.

Chapter 4

Performance Metrics

There are four possible outcomes while using a classifier:

- tp, true positive. The classifier correctly predicts a positive outcome.
- tn, true negative. The classifier correctly predicts a negative outcome.
- fp, false positive. The classifier incorrectly predicts a positive outcome.
- fn, false negative. The classifier incorrectly predicts a negative outcome.

In this dissertation, classifiers were used both for change classification and fix prediction. Thus, it is useful to express performance metrics for both contexts. A classifier's prediction is typically binary. When change classification is performed, a code change is classified as buggy or clean.

During change classification, a true positive is correctly predicting a code change as buggy. A true negative is correctly predicting a clean change. A false positive is when a clean change is predicted as buggy. In practice, this can be quite annoying

for developers. Finally, a false negative is when a classifier predicts a code change to be clean when it is actually buggy. In some fields such as medicine, false negatives are dangerous. Diagnosing a cancer-prone patient as healthy can lead to ethical and legal issues. However, in typical software engineering work, false positives are more troublesome than false negatives. Developers are far less motivated to use a system which points out fake bugs.

The meaning of these outcomes differs for fix suggestion. During this process, a prediction is made on every element of a promising group of code tokens. Each token is classified as present or absent in eventual bug fix. A true positive means that a predicted fix code token is present in the actual bug fix. A true negative means that a token not predicted to be in the fix was absent in the actual fix. A false positive indicates that a token predicted to be in the fix is not in the actual fix. Finally, a false negative indicates that a token not predicted to be in the fix is present in the actual fix.

With a good set of training data, it is possible to compute the number of true positives n_{tp}, true negatives n_{tn}, false positives n_{fp}, and false negatives n_{fn}.

$$Accuracy = \frac{n_{tp} + n_{tn}}{n_{tp} + n_{tn} + n_{fp} + n_{fn}}$$

Accuracy is the number of correct predictions over the total number of predictions. Bug prediction typically deals with more clean changes than buggy changes. Using this measure could yield a high value if clean changes are being better predicted than buggy changes. During fix content prediction, accuracy does not reveal the disparity between false positives and false negatives. Overall, the accuracy figure is often less relevant

than precision and recall.

$$Precision, P = \frac{n_{tp}}{n_{tp} + n_{fp}}$$

For buggy change prediction, it is the number of code changes which are actually buggy when predicted to be buggy. For fix content prediction, this represents the number of correct predictions over the total number of tokens predicted to be in the fix.

$$Recall, R = \frac{n_{tp}}{n_{tp} + n_{fn}}$$

Also known as the true positive rate, this represents the number of correct bug classifications over the total number of changes that were actually bugs. For fix content prediction, it is the likelihood that a token in an actual bug fix was predicted by the classifier.

$$F - measure = \frac{2 * P * R}{P + R}$$

This is a composite measure of buggy change precision and recall, more precisely, it is the harmonic mean of precision and recall. Since precision can often be improved at the expense of recall (and vice-versa), F-measure is a good measure of the overall precision/recall performance of a classifier, since it incorporates both values. For this reason, we emphasize this metric in our analysis of classifier performance in chapters 5 and 6.

An extreme example is when the Fix Suggester predicts just token for a code change. Let's assume this token was correctly predicted. This results in a precision of 100% and a recall of 0%. Predicting all tokens in a vocabulary leads to 100% recall at huge cost to precision. Presenting the F-measure allows one to better understand the precision recall trade-off. Accordingly, chapter 6 relays fix content prediction F-measure throughout project history for all corpus projects. Several data/text mining papers have compared performance on classifiers using F-measure including [7] and [98].

In chapter 5, accuracy is also reported in order to avoid returning artificially higher F-measures when accuracy is low. For example, suppose there are 10 code changes, 5 of which are buggy. If a classifier predicts all changes as buggy, the resulting precision, recall, and F-measure are 50%, 100%, and 66.67% respectively. The accuracy figure of 50% demonstrates less promise than the high F-measure figure suggests.

An ROC (originally, receiver operating characteristic, typically called as ROC) curve is a two-dimensional graph where the true positive rate (i.e. recall, the number of items correctly labeled as belonging to the class) is plotted on the Y axis against the false positive rate (i.e. items incorrectly labeled as belonging to the class, or $n_{c \to b}/(n_{c \to b}+n_{c \to c})$) on the X axis. The area under an ROC curve, commonly abbreviated as ROC AUC, has an important statistical property. The ROC AUC of a classifier is equivalent to the probability that the classifier will value a randomly chosen positive instance higher than a randomly chosen negative instance [56]. These instances map to code changes when relating to bug prediction. Tables 5.4 and 5.5 contain the ROC AUC figures for each project.

For the Fix Suggester, the precision, recall, F-measure metrics refer to the class of code tokens which are predicted to be part of a fix. For the Bug Prognosticator, these metrics refer to changes which are predicted to be buggy. One can also use precision and recall to refer to changes which are predicted to be clean or to predict tokens which are not part of a future fix. For the sake of simplicity, this dissertation refers to precision and recall in the straightforward context of predicting a change to be buggy and predicting fix tokens of a future bug fix.

One potential problem when computing these performance metrics is the choice of which data is used to train and test the classifier. *We consistently use the standard technique of 10-fold cross-validation [5] when computing performance metrics in chapter 5 (for all figures and tables)* with the exception of section 5.7.3 where more extensive cross-fold validation was performed. This avoids the potential problem of over fitting to a particular training and test set within a specific project. In 10-fold cross validation, a data set (i.e., the project revisions) is divided into 10 parts (partitions) at random. One partition is removed and is the target of classification, while the other 9 are used to train the classifier. Classifier evaluation metrics, including buggy and clean accuracy, precision, recall, F-measure, and ROC AUC are computed for each partition. After these evaluations are completed for all 10 partitions, averages are obtained. The classifier performance metrics reported in this chapter are these average figures.

In chapter 6, overfitting was avoided by ensuring that Fix suggestions were always made for future data using past data as training. For example, the first 10% of project history can be used as the train set and the remaining 90% as the test set. A

lot more revisions were gathered for the Fix Suggester corpus. The presence of more historical data allows one to validate sufficiently with large train and test sets.

The next chapter discusses the Bug Prognosticator.

Chapter 5

Bug Prognosticator

5.1 Introduction

Imagine if you had a little imp, sitting on your shoulder, telling you whether your latest code change has a bug. Imps, being mischievous by nature, would occasionally get it wrong, telling you a clean change was buggy. This would be annoying. However, say the imp was right 90% of the time. Would you still listen to him?

While real imps are in short supply, thankfully advanced machine learning classifiers are readily available (and have plenty of impish qualities). Prior work by the second and fourth authors (hereafter called Kim et al. [87]) and similar work by Hata et al. [73] demonstrate that classifiers, when trained on historical software project data, can be used to predict the existence of a bug in an individual file-level software change. The classifier is first trained on information found in historical changes, and once trained, can be used to classify a new impending change as being either buggy

(predicted to have a bug) or clean (predicted to not have a bug).

The traditional model of programming involves many weeks, months, or even years of development before code written by developers is actively tested by quality assurance teams. Once quality assurance teams start testing code, it can additionally take weeks before bugs are discovered. Thus, the turn around time from a bug inducing change, it being discovered by quality assurance, and the bug being fixed can vary between several weeks to several months. An agile development process might reduce the turnaround time to a few weeks [1]. However, this is still a considerable turnaround time for many bugs. Further shortening the time will undoubtedly benefit organizations. Change classification is a technique which can alert the developer within minutes if the code change he/she just committed is buggy. The change classification technique was described fully in chapter 3.

We envision a future where software engineers have bug prediction capability built into their development environment [107]. Instead of an imp on the shoulder, software engineers will receive feedback from a classifier on whether a change they committed is buggy or clean. During this process, a software engineer completes a change to a source code file, submits the change to a software configuration management (SCM) system, then receives a bug prediction back on that change. If the change is predicted to be buggy, a software engineer could perform a range of actions to find the latent bug, including writing more unit tests, performing a code inspection, or examining similar changes made elsewhere in the project.

Due to the need for the classifier to have up-to-date training data, the predic-

tion is performed by a bug prediction service located on a server machine [107]. Since the service is used by many engineers, speed is of the essence when performing bug predictions. Faster bug prediction means better scalability, since quick response times permit a single machine to service many engineers.

A bug prediction service must also provide precise predictions. If engineers are to trust a bug prediction service, it must provide very few “false alarms”, changes that are predicted to be buggy but are really clean [24]. If too many clean changes are falsely predicted to be buggy, developers will lose faith in the bug prediction system.

The prior change classification bug prediction approach used by Kim et al. and analogous work by Hata et al. involve the extraction of “features” (in the machine learning sense, which differ from software features) from the history of changes made to a software project. They include everything separated by whitespace in the code that was added or deleted in a change. Hence, all variables, comment words, operators, method names, and programming language keywords are used as features to train the classifier. Some object-oriented metrics are also used as part of the training set, together with other features such as configuration management log messages and change metadata (size of change, day of change, hour of change, etc.). This leads to a large number of features: in the thousands and low tens of thousands. For project histories which span a thousand revisions or more, this can stretch into hundreds of thousands of features. Changelog features were included as a way to capture the intent behind a code change, for example to scan for bug fixes. Metadata features were included on the assumption that individual developers and code submission times are positively correlated with bugs

[149]. Code complexity metrics were captured due to their effectiveness in previous studies ([120, 46] amongst many others). Source keywords are captured en masse as they contain detailed information for every code change.

The large feature set comes at a cost. Classifiers typically cannot handle such a large feature set, in the presence of complex interactions and noise. For example, the addition of certain features can reduce the accuracy, precision, and recall of a support vector machine. Due to the value of a specific feature not being known a-priori, it is necessary to start with a large feature set and gradually reduce features. Additionally, the time required to perform classification increases with the number of features, rising to several seconds per classification for tens of thousands of features, and minutes for large project histories. This negatively affects the scalability of a bug prediction service.

A possible approach (from the machine learning literature) for handling large feature sets is to perform a feature selection process to identify that subset of features providing the best classification results. A reduced feature set improves the scalability of the classifier, and can often produce substantial improvements in accuracy, precision, and recall.

This chapter investigates multiple feature selection techniques to improve classifier performance. Classifier performance can be evaluated using a suitable metric. For this chapter, classifier performance refers to buggy F-measure rates. The choice of this measure was discussed in chapter 4. The feature selection techniques investigated include both filter and wrapper methods. The best technique is Significance Attribute Evaluation (a filter method) in conjunction with the Naïve Bayes classifier. This tech-

nique discards features with lowest significance until optimal classification performance is reached (described in section 5.5).

Although many classification techniques could be employed, this chapter focuses on the use of Naïve Bayes and SVM. The reason is due to the strong performance of the SVM and the Naïve Bayes classifier for text categorization and numerical data [79, 100]. The J48 and JRIP classifiers were briefly tried, but due to inadequate results, their mention is limited to section 5.9.

The primary contribution of this chapter is the empirical analysis of multiple feature selection techniques to classify bugs in software code changes using file level deltas. An important secondary contribution is the high average F-measure values for predicting bugs in individual software changes.

5.2 Research Questions

This chapter explores the following research questions.

Question 1. Which variables lead to best bug prediction performance when using feature selection?

The three variables affecting bug prediction performance that are explored in this chapter are: (1) type of classifier (Naïve Bayes, Support Vector Machine), (2) type of feature selection used (3) and whether multiple instances of a feature are significant (count), or whether only the existence of a feature is significant (binary). Results are reported in Section 5.7.1.

Results for question 1 are reported as averages across all projects in the corpus. However, in practice it is useful to know the range of results across a set of projects. This leads to the second question.

Question 2. Range of bug prediction performance using feature selection. How do the best-performing SVM and Naïve Bayes classifiers perform across all projects when using feature selection? (See Section 5.7.2)

The sensitivity of bug prediction results with number of features is explored in the next question.

Question 3. Feature Sensitivity. What is the performance of change classification at varying percentages of features? What is the F-measure of the best performing classifier when using just 1% of all project features? (See Section 5.7.4)

Some types of features work better than others for discriminating between buggy and clean changes, explored in the final research question.

Question 4. Best Performing Features. Which classes of features are the most useful for performing bug predictions? (See Section 5.7.5)

A comparison of this chapter's results with those found in related work (see chapter 5.8) shows that change classification with feature selection, the Bug Prognosticator, outperforms other existing classification-based bug prediction approaches. Furthermore, when using the Naïve Bayes classifier, buggy precision averages 97% with a recall of 70%, indicating the predictions are generally highly precise, thereby minimizing the impact of false positives.

In the remainder of the chapter, we begin by presenting a modification to the

Change classification approach of chapter 3. This includes a new process for feature selection. Next, we describe the experimental context, including our data set, and specific classifiers (Section 5.6). The stage is now set, and in subsequent sections we explore the research questions described above (Sections 5.7.1 - 5.7.5). A brief investigation of algorithm runtimes is next (Section 5.7.8). The chapter ends with a comparison with related work (Section 5.8), threats to validity (Section 5.9), and the conclusion.

5.3 Feature Selection

Change classification is applied as per the steps in chapter 3. Most of the steps until this point are the same as in Kim et al [87]. The following step, taking place of old step 5 is the new contribution to change classification in this chapter.

Feature Selection:

Step 5. Perform a feature selection process that employs a combination of wrapper and filter methods to compute a reduced set of features. The filter methods used are Gain Ratio, Chi-Squared, Significance, and Relief-F feature rankers. The wrapper methods are based on the Naïve Bayes and the SVM classifiers. For each iteration of feature selection, classifier F-measure is optimized. As Relief-F is a slower method, it is only invoked on the top 15% of features rather than 50%. Feature selection is iteratively performed until one feature is left. At the end of Step 5, there is a reduced feature set that performs optimally for the chosen classifier metric.

Table 5.1 summarizes features generated and used in this chapter. Feature

groups which can be interpreted as binary or count are also indicated.

Section 5.7.1 explores binary and count interpretations for those feature groups.

Table 5.2 provides an overview of the projects examined in this chapter and the duration of each project examined.

For each project we analyzed, (see Table 5.2) the number of meta-data (M) and code complexity (C) features is 8 and 150 respectively. Source code (A, D, N), change log (L), and directory/filename (F) features contribute thousands of features per project, ranging from 6 to 40 thousand features. Source code features (A, D, N) can take two forms: binary or count. For binary features, the feature only notes the presence or absence of a particular keyword. For count features, the count of the number of times a keyword appears is recorded. For example, if a variable maxParameters is present anywhere in the project, a binary feature just records this variable as being present, while a count feature would additionally record the number of times it appears anywhere in the project's history.

5.4 Feature Selection Techniques

The number of features gathered during the feature extraction phase is quite large, ranging from 6,127 for Plone to 41,942 for JCP (Table 5.2). Such large feature sets lead to longer training and prediction times, require large amounts of memory to perform classification. A common solution to this problem is the process of feature selection, in

Table 5.1: Feature groups. Feature group description, extraction method, and example features.

Feature Group	Description	Extraction Method	Example Features	Feature Interpretation
Added Delta (A)	Terms in added delta source code	BOW+	if, while, for, ==	Binary/Count
Deleted Delta (D)	Terms in deleted delta source code	BOW+	true, 0, <=, ++, int	Binary/Count
Directory/File Name (F)	Terms in directory/file names	BOW++	src, module, java	N/A
Change Log (L)	Terms in the change log	BOW	fix, added, new	N/A
New Revision Source Code (N)	Terms in new Revision source code file	BOW+	if,‖, !=, do, while, string, false	Binary/Count
Meta-data (M)	Change meta-data such as time and author	Direct	author: hunkim, commit hour: 12	N/A
Complexity Metrics (C)	Software complexity metrics of each source code unit	Understand [76] tools	LOC: 34, Cyclomatic: 10	N/A

Table 5.2: Summary of Projects Surveyed

Project	Period	Clean Changes	Buggy Changes	Features
Apache 1.3	10/1996 - 01/1997	566	134	17,575
Columba	05/2003 - 09/2003	1,270	530	17,411
Gaim	08/2000 - 03/2001	742	451	9,281
GForge	01/2003 - 03/2004	399	334	8,996
JEdit	08/2002 - 03/2003	626	377	13,879
Mozilla	08/2003 - 08/2004	395	169	13,648
Eclipse	10/2001 - 11/2001	592	67	16,192
Plone	07/2002 - 02/2003	457	112	6,127
PostgreSQL	11/1996 - 02/1997	853	273	23,247
Subversion	01/2002 - 03/2002	1,925	288	14,856
JCP	1 year	1,516	403	41,942
Total	N/A	9,294	3,125	183,054

which only the subset of features that are most useful for making classification decisions are actually used.

This chapter empirically compares a selection of wrapper and filter methods for bug prediction classification effectiveness. Filter methods use general characteristics of the dataset to evaluate and rank features [70]. They are independent of learning algorithms. Wrapper methods, on the other hand, evaluate features by using scores provided by learning algorithms. All methods can be used with both count and binary interpretations of keyword features.

The methods are further described below.

- *Gain Ratio Attribute Evaluation* - Gain Ratio is a myopic feature scoring algorithm. A myopic feature scoring algorithm evaluates each feature individually independent of the other features. Gain Ratio improves upon Information Gain [5], a well known measure of the amount by which a given feature contributes information to a classification decision. Information Gain for a feature in our context is the amount of information the feature can provide about whether the code change is buggy or clean. Ideally, features that provide a good split between buggy and clean changes should be preserved. For example, if the presence of a certain feature strongly indicates a bug, and its absence greatly decreases the likelihood of a bug, that feature possesses strong Information Gain. On the other hand, if a feature's presence indicates a 50% likelihood of a bug, and its absence also indicates a 50% likelihood of a bug, this feature has low Information Gain

and is not useful in predicting the code change.

However, Information Gain places more importance on features that have a large range of values. This can lead to problems with features such as the LOC of a code change. Gain Ratio [5] plays the same role as Information Gain, but instead provides a normalized measure of a feature's contribution to a classification decision [5]. Thus, Gain Ratio is less affected by features having a large range of values. More details on how the entropy based measure is calculated for Gain Ratio (including how a normalized measure of a feature's contribution is computed), and other inner workings can be found in an introductory data mining book, e.g. [5].

- *Chi-Squared Attribute Evaluation* - Chi-squared feature score is also a myopic feature scoring algorithm. The worth of a feature is given by the value of the Pearson chi-squared statistic [5] with respect to the classification decision. Based on the presence of a feature value across all instances in the dataset, one can compute expected and observed counts using Pearson's chi-squared test. The features with the highest disparity between expected and observed counts against the class attribute are given higher scores.

- *Significance Attribute Evaluation* - With significance attribute scoring, high scores are given to features where an inversion of the feature value (e.g., a programming keyword not being present when it had previously been present) will very likely cause an inversion of the classification decision (buggy to clean, or vice-versa) [4]. It too is a myopic feature scoring algorithm. The significance itself is computed

as a two-way function of its association to the class decision. For each feature, the attribute-to-class association along with the class-to-attribute association is computed. A feature is quite significant if both of these associations are high for a particular feature. More detail on the inner workings can be found in [4].

- *Relief-F Attribute Selection* - Relief-F in an extension to the Relief algorithm [93]. Relief samples data points (code changes in the context of this chapter) at random, and computes two nearest neighbors: one neighbor which has the same class as the instance (similar neighbor), and one neighbor which has a different class (differing neighbor). For the context of this chapter, the 2 classes are buggy and clean. The quality estimation for a feature f is updated based on the value of its similar and differing neighbors. If for feature f, a code change and its similar neighbor have differing values, the feature quality of f is decreased. If a code change and its differing neighbor have different values for f, f's feature quality is increased. This process is repeated for all the sampled points. Relief-F is an algorithmic extension to Relief [135]. One of the extensions is to search for the nearest k neighbors in each class rather than just one neighbor. As Relief-F is a slower algorithm than the other presented filter methods, it was used on the top 15% features returned by the best performing filter method.

- *Wrapper Methods* - The wrapper methods leveraged the classifiers used in the study. The SVM and the Naïve Bayes classifier were used as wrappers. The features are ranked by their classifier computed score. The top feature scores are

those features which are valued highly after the creation of a model driven by the SVM or the Naïve Bayes classifier.

5.5 Feature Selection Process

Filter and wrapper methods are used in an iterative process of selecting incrementally smaller sets of features, as detailed in Algorithm 1. The process begins by cutting the initial feature set in half, to reduce memory and processing requirements for the remainder of the process. The process performs optimally when under 10% of all features are present. The initial feature evaluations for filter methods are 10-fold cross validated in order to avoid the possibility of over-fitting feature scores. As wrapper methods perform 10-fold cross validation during the training process, feature scores from wrapper methods can directly be used. Note that we cannot stop the process around the 10% mark and assume the performance is optimal. Performance may not be a unimodal curve with a single maxima around the 10% mark, it is possible that performance is even better at, say, the 5% mark.

In the iteration stage, each step finds those 50% of remaining features that are least useful for classification using individual feature rank, and eliminates them (if, instead, we were to reduce by one feature at a time, this step would be similar to backward feature selection [104]). Using 50% of features at a time improves speed without sacrificing much result quality.

So, for example, $selF$ starts at 50% of all features, then is reduced to 25% of

all features, then to 12.5%, and so on. At each step, change classification bug prediction using $selF$ is then performed over the entire revision history, using 10-fold cross validation to reduce the possibility of over-fitting to the data.

This iteration terminates when only one of the original features is left. At this point, there is a list of tuples: feature %, feature selection technique, classifier performance. The final step involves a pass over this list to find the feature % at which a specific classifier achieves its greatest performance. The metric used in this chapter for classifier performance is the buggy F-measure (harmonic mean of precision and recall), though one could use a different metric.

It should be noted that algorithm 1 is itself a wrapper method as it builds an SVM or Naïve Bayes classifier at various points. When the number of features for a project is large, the learning process can take a long time. This is still strongly preferable to a straightforward use of backward feature selection, which removes one feature every iteration. An analysis on the runtime of the feature selection process and its components is performed in section 5.7.8. When working with extremely large datasets, using algorithm 1 exclusively with filter methods will also significantly lower its runtime.

5.6 Experimental Context

We gathered software revision history for Apache, Columba, Gaim, Gforge, Jedit, Mozilla, Eclipse, Plone, PostgreSQL, Subversion, and a commercial project writ-

ten in Java (JCP). These are all mature open source projects with the exception of JCP. In this chapter, these projects are collectively called the corpus.

Using the project's CVS (Concurrent Versioning System) or SVN (Subversion) source code repositories, we collected revisions 500-1000 for each project, excepting Jedit, Eclipse, and JCP. For Jedit and Eclipse, revisions 500-750 were collected. For JCP, a year's worth of changes were collected. We used the same revisions as Kim et al. [87], since we wanted to be able to compare our results with theirs. We removed two of the projects they surveyed from our analysis, Bugzilla and Scarab, as the bug tracking systems for these projects did not distinguish between new features and bug fixes. Even though our technique did well on those two projects, the value of bug prediction when new features are also treated as bug fixes is arguably less meaningful.

5.7 Results

The following sections present results obtained when exploring the four research questions. For the convenience of the reader, each result section repeats the research question that will be addressed.

5.7.1 Classifier performance comparison

Research Question 1. Which variables lead to best bug prediction performance when using feature selection?

The three main variables affecting bug prediction performance that are explored in this chapter are: (1) type of classifier (Naïve Bayes, Support Vector Machine),

(2) type of feature selection used (3) and whether multiple instances of a particular feature are significant (count), or whether only the existence of a feature is significant (binary). For example, if a variable by the name of "maxParameters" is referenced four times during a code change, a binary feature interpretation records this variable as 1 for that code change, while a count feature interpretation would record it as 4.

The permutations of variables 1, 2, and 3 are explored across all 11 projects in the data set with the best performing feature selection technique for each classifier. For SVMs, a linear kernel with optimized values for C is used. C is a parameter that allows one to trade off training error and model complexity. A low C tolerates a higher number of errors in training, while a large C allows fewer train errors but increases the complexity of the model. [5] covers the SVM C parameter in more detail.

For each project, feature selection is performed, followed by computation of per-project accuracy, buggy precision, buggy recall, and buggy F-measure. Once all projects are complete, average values across all projects are computed. Results are reported in Table 5.3. Average numbers may not provide enough information on the variance. Every result in Table 5.3 reports the average value along with the standard deviation. The results are presented in descending order of buggy F-measure.

Significance Attribute and Gain Ratio based feature selection performed best on average across all projects in the corpus when keywords were interpreted as binary features. McCallum et al. [111] confirm that feature selection performs very well on sparse data used for text classification. Anagostopoulos et al. [7] report success with Information Gain for sparse text classification. Our data set is also quite sparse and

Table 5.3: Average Classifier Performance on Corpus (ordered by descending Buggy F-measure)

Feature Interp.	Classifier	Feature Selection Technique	Features	Feature Percentage	Accuracy	Buggy F-measure	Buggy Precision	Buggy Recall
Binary	Bayes	Significance Attr.	1153.4 ± 756.3	7.9 ± 6.3	0.90 ± 0.04	0.81 ± 0.07	0.97 ± 0.06	0.70 ± 0.10
Binary	Bayes	Gain Ratio	1914.6 ± 2824.0	9.5 ± 8.0	0.91 ± 0.03	0.79 ± 0.07	0.92 ± 0.10	0.70 ± 0.07
Binary	Bayes	Relief-F	1606.7 ± 1329.2	9.3 ± 3.6	0.85 ± 0.08	0.71 ± 0.10	0.74 ± 0.19	0.71 ± 0.08
Binary	SVM	Gain Ratio	1522.3 ± 1444.2	10.0 ± 9.8	0.87 ± 0.05	0.69 ± 0.06	0.90 ± 0.10	0.57 ± 0.08
Binary	Bayes	Wrapper	2154.6 ± 1367.0	12.7 ± 6.5	0.85 ± 0.06	0.68 ± 0.10	0.76 ± 0.17	0.64 ± 0.11
Binary	SVM	Significance Attr.	1480.9 ± 2368.0	9.5 ± 14.2	0.86 ± 0.06	0.65 ± 0.06	0.94 ± 0.11	0.51 ± 0.08
Count	Bayes	Relief-F	1811.1 ± 1276.7	11.8 ± 1.9	0.78 ± 0.07	0.63 ± 0.08	0.56 ± 0.09	0.72 ± 0.10
Binary	SVM	Relief-F	663.1 ± 503.2	4.2 ± 3.5	0.83 ± 0.04	0.62 ± 0.06	0.75 ± 0.22	0.57 ± 0.14
Binary	Bayes	Chi-Squared	3337.2 ± 2356.1	25.0 ± 20.5	0.77 ± 0.08	0.61 ± 0.07	0.54 ± 0.07	0.71 ± 0.09
Count	SVM	Relief-F	988.6 ± 1393.8	5.8 ± 4.0	0.78 ± 0.08	0.61 ± 0.06	0.58 ± 0.10	0.66 ± 0.07
Count	SVM	Gain Ratio	1726.0 ± 1171.3	12.2 ± 8.3	0.80 ± 0.07	0.57 ± 0.08	0.62 ± 0.13	0.53 ± 0.08
Binary	SVM	Chi-Squared	2561.5 ± 3044.1	12.6 ± 16.2	0.81 ± 0.06	0.56 ± 0.09	0.55 ± 0.11	0.57 ± 0.07
Count	SVM	Significance Attr.	1953.0 ± 2416.8	12.0 ± 13.6	0.79 ± 0.06	0.56 ± 0.07	0.62 ± 0.13	0.52 ± 0.06
Count	SVM	Wrapper	1232.1 ± 1649.7	8.9 ± 14.9	0.76 ± 0.10	0.55 ± 0.05	0.56 ± 0.09	0.56 ± 0.07
Count	Bayes	Wrapper	1946.6 ± 6023.6	4.8 ± 14.3	0.79 ± 0.08	0.55 ± 0.08	0.63 ± 0.08	0.50 ± 0.09
Count	Bayes	Chi-Squared	58.3 ± 59.7	0.4 ± 0.4	0.76 ± 0.08	0.55 ± 0.05	0.54 ± 0.09	0.58 ± 0.11
Count	Bayes	Significance Attr.	2542.1 ± 1787.1	14.8 ± 6.6	0.76 ± 0.08	0.54 ± 0.07	0.53 ± 0.11	0.59 ± 0.13
Count	SVM	Chi-Squared	4282.3 ± 3554.2	26.2 ± 20.2	0.75 ± 0.10	0.54 ± 0.07	0.55 ± 0.10	0.54 ± 0.06
Binary	SVM	Wrapper	4771.0 ± 3439.7	33.8 ± 22.8	0.79 ± 0.11	0.53 ± 0.12	0.58 ± 0.14	0.49 ± 0.11
Count	Bayes	Gain Ratio	2523.6 ± 1587.2	15.3 ± 6.1	0.77 ± 0.08	0.53 ± 0.06	0.56 ± 0.10	0.52 ± 0.09

performs well using Gain Ratio, an improved version of Information Gain. A sparse data set is one in which instances contain a majority of zeroes. The number of distinct ones for most of the binary attributes analyzed are a small minority. The least sparse binary features in the datasets have non zero values in the 10% - 15% range.

Significance attribute based feature selection narrowly won over Gain Ratio in combination with the Naïve Bayes classifier. With the SVM classifier, Gain Ratio feature selection worked best. Filter methods did better than wrapper methods. This is due to the fact that classifier models with a lot of features are worse performing and can drop the wrong features at an early cutting stage of the feature selection process presented under algorithm 1. When a lot of features are present, the data contains more outliers. The worse performance of Chi-Squared Attribute Evaluation can be attributed to computing variances at the early stages of feature selection. Relief-F is a nearest neighbor based method. Using nearest neighbor information in the presence of noise can be detrimental [60]. However, as Relief-F was used on the top 15% of features returned by the best filter method, it performed reasonably well.

It appears that simple filter based feature selection techniques such as Gain Ratio and Significance Attribute Evaluation can work on the surveyed large feature datasets. These techniques do not make strong assumptions on the data. When the amount of features is reduced to a reasonable size, wrapper methods can produce a model usable for bug prediction.

Overall, Naïve Bayes using binary interpretation of features performed better than the best SVM technique. One explanation for these results is that change classi-

fication is a type of optimized binary text classification, and several sources ([7, 111]) note that Naïve Bayes performs text classification well.

As a linear SVM was used, it is possible that using a non linear kernel can improve SVM classifier performance. However, in preliminary work, optimizing the SVM for one project by using a polynomial kernel often led to degradation of performance in another project. In addition, using a non linear kernel, made the experimental runs significantly slower. To permit a consistent comparison of results between SVM and Naïve Bayes, it was important to use the same classifier settings for all projects, and hence no per-project SVM classifier optimization was performed. Future work could include optimizing SVMs on a per project basis for better performance. A major benefit of the Naïve Bayes classifier is that such optimization is not needed.

The methodology section discussed the differences between binary and count interpretation of keyword features. Count did not perform well within the corpus, yielding poor results for SVM and Naïve Bayes classifiers. Count results are mostly bundled at the bottom of Table 5.3. This is consistent with McCallum et al. [111], which mentions that when the number of features is low, binary values can yield better results than using count. Even when the corpus was tested without any feature selection in the prior work by Kim et al., count performed worse than binary. This seems to imply that regardless of the number of features, code changes are better captured by the presence or absence of keywords. The bad performance of count can possibly be explained by the difficulty in establishing the semantic meaning behind recurrence of keywords and the added complexity it brings to the data. When using the count interpretation, the

top results for both Naïve Bayes and the SVM were from Relief-F. This suggests that using Relief-F after trimming out most features via a myopic filter method can yield good performance.

The two best performing classifier combinations by buggy F-measure, Bayes (binary interpretation with Significance Evaluation) and SVM (binary interpretation with Gain Ratio), both yield an average precision of 90% and above. For the remainder of the chapter, analysis focuses just on these two, to better understand their characteristics. The remaining figures and tables in the chapter will use a binary interpretation of keyword features.

5.7.2 Effect of feature selection

Question 2. Range of bug prediction performance using feature selection. How do the best-performing SVM and Naïve Bayes classifiers perform across all projects when using feature selection?

In the previous section, aggregate average performance of different classifiers and optimization combinations was compared across all projects. In actual practice, change classification would be trained and employed on a specific project. As a result, it is useful to understand the range of performance achieved using change classification with a reduced feature set. Tables 5.4 and 5.5 below report, for each project, overall prediction accuracy, buggy and clean precision, recall, F-measure, and ROC area under curve (AUC). Table 5.4 presents results for Naïve Bayes using F-measure feature selection with binary features, while Table 5.5 presents results for SVM using feature

selection with binary features.

Observing these two tables, eleven projects overall (8 with Naïve Bayes, 3 with SVM) achieve a buggy precision of 1, indicating that all buggy predictions are correct (no buggy false positives). While the buggy recall figures (ranging from 0.40 to 0.84) indicate that not all bugs are predicted, still, on average, more than half of all project bugs are successfully predicted.

Comparing buggy and clean F-measures for the two classifiers, Naïve Bayes (binary) clearly outperforms SVM (binary) across all 11 projects when using the same feature selection technique. The ROC AUC figures are also better for the Naïve Bayes classifier than those of the SVM classifier across all projects. The ROC AUC of a classifier is equivalent to the probability that the classifier will value a randomly chosen positive instance higher than a randomly chosen negative instance. A higher ROC AUC for the Naïve Bayes classifier better indicates that the classifier will still perform strongly if the initial labeling of code changes as buggy/clean turned out to be incorrect. Thus, the Naïve Bayes results are less sensitive to inaccuracies in the data sets. [5].

Figure 5.1 summarizes the relative performance of the two classifiers and compares against the prior work of Kim et al [87]. Examining these figures, it is clear that feature selection significantly improves F-measure of bug prediction using change classification. As precision can often be increased at the cost of recall and vice-versa, we compared classifiers using buggy F-measure. Good F-measures indicate overall result quality.

The practical benefit of our result can be demonstrated by the following. For

a given fixed level of recall, say 70%, our method provides 97% precision, while Kim et al. provide 52.5% precision. These numbers were calculated using buggy F-measure numbers from 5.4 and [87]. The buggy precision value was extrapolated from the buggy F-measure and the fixed recall figures. High precision with decent recall allows developers to use our solution with added confidence. This means that if a change is flagged as buggy, the change is very likely to be actually be buggy.

Kim et al.'s results, shown in Figure 5.1, are taken from [87]. In all but the following cases, this chapter used the same corpus.

- This chapter does not use Scarab and Bugzilla because those projects did not distinguish buggy and new features.

- This chapter uses JCP, which was not in Kim et al.'s corpus.

Kim et al. did not perform feature selection and used substantially more features for each project. Table 5.3 reveals the drastic reduction in the average number of features per project when compared to the paper by Kim et al.

Additional benefits of the reduced feature set include better speeds of classification and scalability. As linear SVMs and Naïve Bayes classifiers work in linear time for classification [80], removing 90% of features allows classifications to be done about ten times faster. We have noted an order of magnitude improvement in code change classification time. This helps promote interactive use of the system within an Integrated Development Environment.

5.7.3 Statistical Analysis of Results

While the above results appear compelling, it is necessary to further analyze them to be sure that the results did not occur due to statistical anomalies. There are a few possible sources of anomaly.

- The 10-fold validation is an averaging of the results for F-measure, accuracy, and all of the stats presented. It is possible that the variance of the cross validation folds is high.
- Statistical analysis is needed to confirm that a better performing classifier is indeed better after taking into account cross validation variance.

To ease comparison of results against Kim et al. [87], the cross validation random seed was set to the same value used by them. With the same datasets (except for JCP) and the same cross validation random seed, but with far better buggy F-measure and accuracy results, the improvement over no feature selection is straightforward to demonstrate.

Showing that the Naïve Bayes results are better than the SVM results when comparing tables 5.4, 5.5 is a more involved process that is outlined below. For the steps below, the better and worse classifiers are denoted respectively as c_b and c_w.

1. Increase the cross validation cycle to 100 runs of 10-fold validation, with the seed being varied each run.
2. For each run, note the metric of interest, buggy F-measure or accuracy. This

process empirically generates a distribution for the metric of interest, dm_{cb} using the better classifier c_b.

3. Repeat steps 1, 2 for the worse classifier c_w, attaining dm_{cw}.

4. Use a one-sided Kolmogorov Smirnov test to show that the population CDF of distribution dm_{cb} is larger than the CDF of dm_{cw} at the 95% confidence level.

The seed is varied every run in order to change elements of the train and test sets. Note that the seed for each run is the same for both c_w and c_b.

In step four above, one can consider using a two sample bootstrap test to show that the points of dm_{cb} come from a distribution with a higher mean than the points of dm_{cw} [51]. However, the variance within 10-fold validation is the reason for the statistical analysis. The Birnbaum-Tingey one-sided contour analysis was used for the one-sided Kolmogorov-Smirnov test. It takes variance into account and can indicate if the CDF of one set of samples is larger than the CDF of the other set [29, 110]. It also returns a p-value for the assertion. The 95% confidence level was used.

Incidentally, step 1 is also similar to bootstrapping but uses further runs of cross validation to generate more data points. The goal is to ensure that the results of average error estimation via k-fold validation are not curtailed due to variance. Many more cross validation runs help generate an empirical distribution.

This test was performed on every dataset's buggy F-measure and accuracy to compare the performance of the Naïve Bayes classifier to the SVM. In most of the tests, the Naïve Bayes dominated the SVM results in both accuracy and F-measure at

$p < 0.001$. A notable exception is the accuracy metric for the Gforge project.

While tables 5.4 and 5.5 show that the Naïve Bayes classifier has a 1% higher accuracy for Gforge, the empirical distribution for Gforge's accuracy indicates that this is true only at a p of 0.67, meaning that this is far from a statistically significant result. The other projects and the F-measure results for Gforge demonstrate the dominance of the Naïve Bayes results over the SVM.

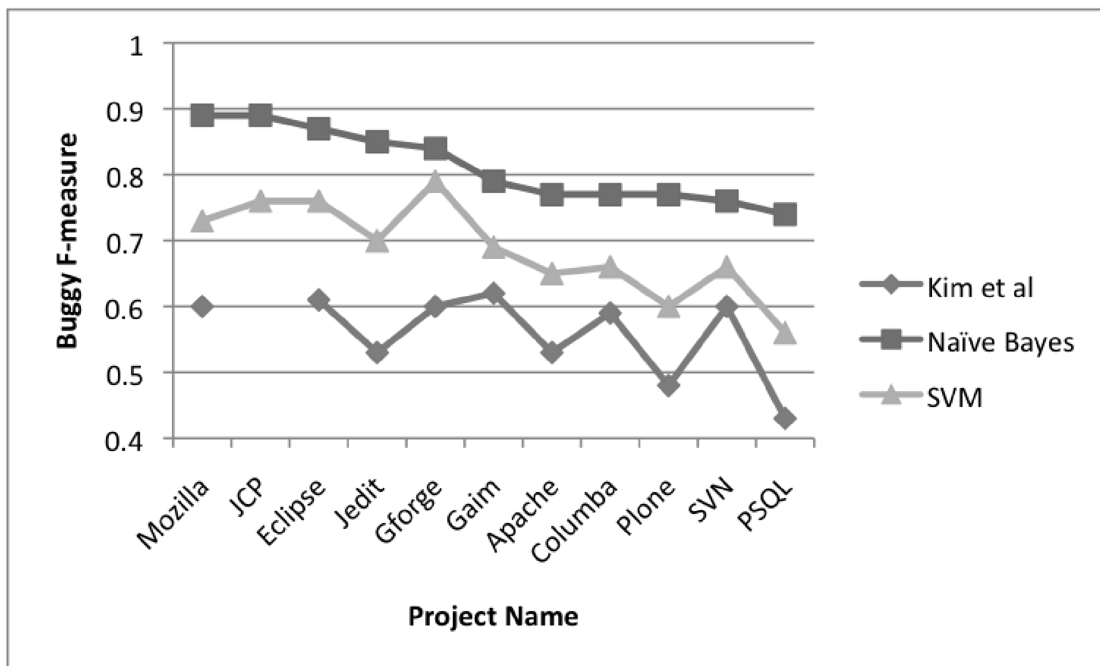
In spite of the results of tables 5.3 and 5.4, it is not possible to confidently state that a binary representation performs better than count for both the Naïve Bayes and SVM classifiers without performing a statistical analysis. Naïve Bayes using binary features dominates over the performance of Naïve Bayes with count features at a $p < 0.001$. The binary SVM's dominance over the best performing count SVM with the same feature selection technique (Gain Ratio) is also apparent, with a $p < 0.001$ on the accuracy and buggy F-measure for most projects, but lacking statistical significance on the buggy F-measure of Gaim.

5.7.4 Feature Sensitivity

Research Question 3. Feature Sensitivity. What is the performance of change classification at varying percentages of features? What is the F-measure of the best performing classifier when using just 1% of all project features?

Section 5.7.2 reported results from using the best feature set chosen using a given optimization criteria, and showed that Naïve Bayes with Significance Attribute feature selection performed best with 7.95% of the original feature set, and SVM with

Figure 5.1: Classifier F-measure by Project



Gain Ratio feature selection performed best at 10.08%. This is a useful result, since the reduced feature set decreases prediction time as compared to Kim et al. A buggy/clean decision is based on about a tenth of the initial number of features. This raises the question of how performance of change classification behaves with varying percentages of features.

To explore this question, for each project we ran a modified version of the feature selection process described in Algorithm 1, in which only 10% (instead of 50%) of features are removed each iteration using Significance Attribute Evaluation. After each feature selection step, buggy F-measure is computed using a Naïve Bayes classifier.

The graph of buggy F-measure versus features, figure 5.2, follows a similar

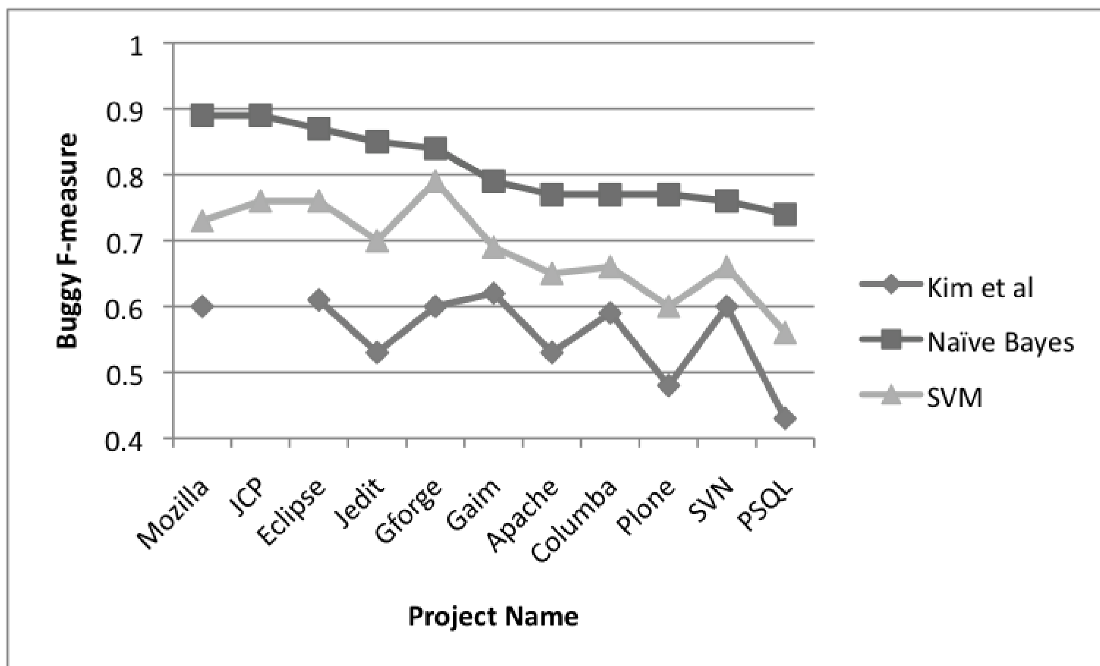
pattern for all projects. As the number of features decrease, the curve shows a steady increase in buggy F-measure. Some projects temporarily dip before an increase in F-measure occurs. Performance can increase with fewer features due to noise reduction but can also decrease with fewer features due to important features missing. The dip in F-measure that typically reverts to higher F-measure can be explained by the presence of correlated features which are removed in later iterations. While correlated features are present at the early stages of feature selection, their influence is limited by the presence of a large number of features. They have a more destructive effect towards the middle before being removed.

Following the curve in the direction of fewer features, most projects then stay at or near their maximum F-measure down to single digit percentages of features. This is significant, since it suggests a small number of features might produce results that are close to those obtained using more features. The reason can be attributed to Menzies et al. [113] who state that a small number of features with different types of information can sometimes capture more information than many features of the same type. In the experiments of the current chapter there are many feature types. Fewer features bring two benefits: a reduction in memory footprint and an increase in classification speed.

A notable exception is the Gforge project. When trimming by 50% percent of features at a time, Gforge's optimal F-measure point (at around 35%) was missed. Feature selection is still useful for Gforge but the optimal point seems to be a bit higher than for the other projects.

In practice, one will not know a-priori the best percentage of features to use

Figure 5.2: Buggy F-measure versus Features using Naïve Bayes



for bug prediction. Empirically from figure 5.2, a good starting point for bug prediction is at 15% of the total number of features for a project. One can certainly locate counterexamples including Gforge from this chapter. Nevertheless, 15% of features is a reasonable practical recommendation if no additional information is available for a project.

To better characterize performance at low numbers of features, accuracy, buggy precision, and buggy recall are computed for all projects using just 1% of features (selected using the Significance Attribute Evaluation process). Results are presented in Table 5.6. When using 1% percent of overall features, it is still possible to achieve high buggy precision, but at the cost of somewhat lower recall. Taking Apache as an

example, using the F-measure optimal number of project features (6.25%, 1098 total) achieves buggy precision of 0.99 and buggy recall of 0.63 (Table 5.4), while using 1% of all features yields buggy precision of 1 and buggy recall of 0.46 (Table 5.6).

These results indicate that a small number of features have strong predictive power for distinguishing between buggy and clean changes. An avenue of future work is to explore the degree to which these features have a causal relationship with bugs. If a strong causal relationship exists, it might be possible to use these features as input to a static analysis of a software project. Static analysis violations spanning those keywords can then be prioritized for human review.

A natural question that follows is the breakdown of the top features. A software developer might ask which code attributes are the most effective predictors of bugs. The next section deals with an analysis of the top 100 features of each project.

5.7.5 Breakdown of Top 100 Features

Research Question 4. Best Performing Features. Which classes of features are the most useful for performing bug predictions?

Table 5.7 provides a breakdown by category of the 100 most prominent features in each project. The top three types are purely keyword related. Adding further occurrences of a keyword to a file has the highest chance of creating a bug, followed by deleting a keyword, followed finally by introducing an entirely new keyword to a file. Changelog features are the next most prominent set of features. These are features based on the changelog comments for a particular code change. Finally, filename

features are present within the top 100 for a few of the projects. These features are generated from the actual names of files in a project.

It is somewhat surprising that complexity features do not make the top 100 feature list. The Apache project has 3 complexity features in the top 100 list. This was not presented in Table 5.7 as Apache is the only project where complexity metrics proved useful. Incidentally, the top individual feature for the Apache project is the average length of the comment lines in a code change.

The absence of metadata features in the top 100 list is also surprising. Author name, commit hour, and commit day do not seem to be significant in predicting bugs. The commit hour and commit day were converted to binary form before classifier analysis. It is possible that encoding new features using a combination of commit hour and day such as "Mornings", "Evenings", or "Weekday Nights" might provide more bug prediction value. An example would be if the hour 22 is not a good predictor of buggy code changes but hours 20-24 are jointly a strong predictor of buggy changes. In this case, encoding "Nights" as hours 20-24 would yield a new feature that is more effective at predicting buggy code changes.

Five of the eleven projects had BOW+ features among the top features including Gforge, JCP, Plone, PSQL, and SVN. The Naïve Bayes optimized feature (Table 5.4) contains BOW+ features for those projects with the addition of Gaim and Mozilla, a total of seven out of eleven projects. BOW+ features help add predictive value.

5.7.6 Alternative to 10-fold Validation

The tables presented in this chapter report results after 10-fold cross validation was performed. An issue with 10-fold cross validation is that code changes are treated as time independent. When reporting 10-fold metrics, the performance numbers might potentially be inflated as future code changes could have be used to predict past changes during a fold of 10-fold validation. This happens because changes are assumed to not exhibit strong time characteristics and the content of each validation fold is randomized. It is also possible that learning from a randomized subset of future changes will not significantly impact results.

This section introduces a temporal equivalent of 10-fold validation. The data is split into 10 chunks. The first chunk has the first 10% of code changes, the next chunk has the next 20% etc. To attain a result similar to 10-fold, the first chunk is used as training, and some data from the second chunk is used as test. The amount of data used from the second chunk is calibrated such that the train data from the first chunk is constitutes 90% of the data, and the test data constitutes 10% of the data. This is repeated for each chunk. The results will be comparable to the traditional 10-fold validation, and can be thought of as 'temporal 10-fold validation'. In order to balance results, averaged Buggy F-measure results of temporal 2-fold validation thru temporal 10-fold validation are presented for each project in table 5.8. This table operates on the same limited feature data sets as table 5.4.

While the results of table 5.8 are slightly worse on buggy F-measure (four

percentage points less on average) compared to table 5.4, they are still quite competitive. In fact, the results can easily be further improved if the feature selection process optimized results for temporal validation as opposed to traditional 10-fold validation. Nevertheless, table 5.8 demonstrates that the bug prognosticator solution can perform well if one is strictly time sensitive.

5.7.7 Alternatives to using the Standard Buggy F-measure

Standard usage of F-measure implies equally valuing precision and recall. In practice, developers often value precision more than recall. Table 5.9 shows the adjusted F-measure if one values precision and recall differently. The $F_{0.5}$ column refers to an F-measure which values recall half as much as precision (or values precision twice as much as recall). The F_2 column refers to a metric that values recall twice as much as precision. The general formula for weighted F-measure when one values recall β times more then precision is

$$Weighted\ F - measure = \frac{(1+\beta)^2 * (precision * recall)}{\beta^2 * precision + recall}[154]$$

The results of table 5.9 show that the optimized results perform better if one values precision more than recall. This is usually the case in practice. Developers normally want to review code changes only when there is strong likelihood of the code change being buggy. They are unlikely to want to re-review a lot of code for the sake of uncovering a small bug. This would practically be akin to intensively searching for a needle in a haystack.

In the event that one values recall more than precision, the result quality degrades. However, this can be rectified during the feature selection process. Instead of aiming for optimize for F-measure, one can can instead optimize classifiers for a weighted F-measure that values recall more than precision to improve the results of table 5.9.

Thus, if one knows the financial cost of a false positive and the cost of a false negative for a particular project, it is possible to figure out a weighted F-measure based on those values. For example, if a false positive costs two dollars and a false negative costs one dollar, one should aim to optimize on the $F_{0.5}$ measure when using the Bug Prognosticator. Interestingly, in such a scenario, it is possible to compare the utility of the classifier to not using one. To do so, one computes the cost of bugs missed when not using a classifier. To get a dollar figure, this is just the number of bugs missed multiplied by two. When using the classifier, one computes the number of incorrectly flagged buggy changes multiplied by one added by the number of missed bugs multiplied by two, to get a comparable dollar figure. In practice, it is hard to gauge a simple flat cost figure for false positives and negatives.

5.7.8 Algorithm Runtime Analysis

The classifiers used in the study consist of linear SVM and Naïve Bayes. The theoretical training time for these classifiers has been analyzed in the research literature. Traditionally, Naïve Bayes is faster to train than a linear SVM. The Naïve Bayes classifier has a training runtime of $O(s*n)$ where s is the number of non-zero features and n is the number of training instances. The SVM traditionally has a runtime of approximately

$O(s*n^{2.1})$ [80]. There is recent work in reducing the runtime of linear SVMs to $O(s*n)$ [80].

Optimizing the classifiers for F-measure slows both down. In our experiments, training an F-measure optimized Naïve Bayes classifier was faster than doing so for the SVM though the speeds were comparable when the number of features are low. The current chapter uses Liblinear [55], one of the fastest implementations of a linear SVM. If a slower SVM package were used, the performance gap would be far wider. The change classification time for both classifiers is linear with the number of features returned after the training.

An important issue is the speed of classification on the projects without any feature selection. While the SVM or Naïve Bayes algorithms do not crash in the presence of a lot of data, training times and memory requirements are considerable. Feature selection allows one to reduce classification time. The time gain on classification is an order of magnitude improvement, as less than 10% of features are typically needed for improved classification performance. This allows individual code classifications to be scalable.

A rough wall-clock analysis of the time required to classify a single code change improved from a few seconds to a few hundred milliseconds when about 10% of features are used (with a 2.26 Ghz Intel Core 2 Duo CPU and 8GB of RAM). Lowered RAM requirements allow multiple trained classifiers to operate simultaneously without exceeding the amount of physical RAM present in a change classification server. The combined impact of reduced classifier memory footprint and reduced classification time will per-

mit a server-based classification service to support substantially more classifications per user.

The top performing filter methods themselves are quite fast. Significance Attribute Evaluation and Gain Ratio also operate linearly with respect to the number of training instances multiplied by the number of features. Rough wall-clock times show 4-5 seconds for the feature ranking process (with a 2.26 Ghz Intel Core 2 Duo CPU and 8GB of RAM). It is only necessary to do this computation once when using these techniques within the feature selection process, algorithm 1. To return the top 5% of features, the entire process takes about 30 seconds.

5.8 Comparison to Related Work

Change classification and faulty program unit detection techniques both aim to locate software defects, but differ in scope and resolution. While change classification focuses on changed entities, faulty module detection techniques do not need a changed entity. The pros of change classification include the following:

- Bug pinpointing at a low level of granularity, typically about 20 lines of code.
- Possibility of IDE integration, enabling a prediction immediately after code is committed.
- Understandability of buggy code prediction from the perspective of a developer.

The cons include the following:

- Excess of features to account for when including keywords.
- Failure to address defects not related to recent file changes.
- Inability to organize a set of modules by likelihood of being defective

Using decision trees and neural networks that employ object-oriented metrics as features, Gyimothy et al. [69] predict fault classes of the Mozilla project across several releases. Their buggy precision and recall are both about 70%, resulting in a buggy F-measure of 70%. Our buggy precision for the Mozilla project is around 100% (+30%) and recall is at 80% (+10%), resulting in a buggy F-measure of 89% (+19%). In addition they predict faults at the class level of granularity (typically by file), while our level of granularity is by code change.

Aversano et al. [18] achieve 59% buggy precision and recall using KNN (K nearest neighbors) to locate faulty modules. Hata et al. [73] show that a technique used for spam filtering of emails can be successfully used on software modules to classify software as buggy or clean. However, they classify static code (such as the current contents of a file), while our approach classifies file changes. They achieve 63.9% precision, 79.8% recall, and 71% buggy F-measure on the best data points of source code history for 2 Eclipse plugins. We obtain buggy precision, recall, and F-measure figures of 100% (+36.1%), 78% (-1.8%) and 87% (+16%), respectively, with our best performing technique on the Eclipse project (Table 5.4). Menzies et al. [113] achieve good results on their best projects. However, their average precision is low, ranging from a minimum of 2% and a median of 20% to a max of 70%. As mentioned in chapter 4, to avoid op-

timizing on precision or recall, we present F-measure figures. A commonality we share with the work of Menzies et al. is their use of Information Gain (quite similar to the Gain Ratio that we use, as explained in section 5.4) to rank features. Both Menzies and Hata focus on the file level of granularity.

Kim et al. show that using support vector machines on software revision history information can provide an average bug prediction accuracy of 78%, a buggy F-measure of 60%, and a precision and recall of 60% when tested on twelve open source projects [87]. Our corresponding results are an accuracy of 92% (+14%), a buggy F-measure of 81% (+21%), a precision of 97% (+37%), and a recall of 70% (+10%). Elish and Elish [53] also used SVMs to predict buggy modules in software. Table 5.10 compares our results with that of earlier work. Hata, Aversano, and Menzies did not report overall accuracy in their results and focused on precision and recall results.

Recently, D'Ambros et al. [46] provided an extensive comparison of various bug prediction algorithms that operate at the file level using ROC AUC to compare algorithms. Wrapper Subset Evaluation is used sequentially to trim attribute size. They find that the top ROC AUC of 0.921 was achieved for the Eclipse project using the prediction approach of Moser et al. [120]. As a comparison, the results achieved using feature selection and change classification in the current chapter achieved an ROC AUC for Eclipse of 0.94. While it's not a significant difference, it does show that change classification after feature selection can provide results comparable to those of Moser et al. which are based on code metrics such as code churn, past bugs, refactorings, number of authors, file age, etc. An advantage of our change classification approach over that

of Moser et al. is that it operates at the granularity of code changes, which permits developers to receive faster feedback on small regions of code. As well, since the top features are keywords (Section 5.7.5), it is easier to explain to developers the reason for a code change being diagnosed as buggy with a keyword diagnosis instead of providing metrics. It is easier to understand a code change being buggy due to having certain combinations of keywords instead of combination of source code metrics.

The next section moves on to work focusing on feature selection.

5.8.1 Feature Selection

Hall and Holmes [70] compare six different feature selection techniques when using the Naïve Bayes and the C4.5 classifier [132]. Each dataset analyzed has about one hundred features. The method and analysis based on iterative feature selection used in this chapter is different from that of Hall and Holmes in that the present work involves substantially more features, coming from a different type of corpus (features coming from software projects). Many of the feature selection techniques used by Hall and Holmes are used in this chapter.

Song et al. [150] propose a general defect prediction framework involving a data preprocessor, feature selection, and learning algorithms. They also note that small changes to data representation can have a major impact on the results of feature selection and defect prediction. This chapter uses a different experimental setup due to the large number of features. However, some elements of this chapter were adopted from Song et al. including cross validation during the feature ranking process for filter

methods. In addition, the Naïve Bayes classifier was used, and the J48 classifier was attempted. The results for the latter are mentioned in Section 7.5. Their suggestion of using a log pre-processor is hard to adopt for keyword changes and cannot be done on binary interpretations of keyword features. Forward and backward feature selection one attribute at a time is not a practical solution for large datasets.

Gao et al. [62] apply several feature selection algorithms to predict defective software modules for a large legacy telecommunications software system. Seven filter-based techniques and three subset selection search algorithms are employed. Removing 85% of software metric features does not adversely affect results, and in some cases improved results. The current chapter uses keyword information as features instead of product, process, and execution metrics. The current chapter also uses historical data. Finally, 11 systems are examined instead of 4 when compared to that work. However similar to that work, a variety of feature selection techniques is used as well on a far larger set of attributes.

5.9 Threats to Validity

There are six major threats to the validity of this study.

Systems examined might not be representative of typical projects. Eleven systems are examined, a quite high number compared to other work reported in literature. In spite of this, it is still possible that we accidentally chose systems that have better (or worse) than average bug classification accuracy. Since we intentionally

chose systems that had some degree of linkage between change tracking systems and change log text (to determine fix inducing changes), there is a project selection bias.

Systems are mostly open source.

The systems examined in this dissertation mostly all use an open source development methodology with the exception of JCP, and hence might not be representative of typical development contexts, potentially affecting external validity [165]. It is possible that more deadline pressures, differing personnel turnover patterns, and varied development processes used in commercial development could lead to different buggy change patterns.

Bug fix data is incomplete.

Even though we selected projects that have decent change logs, we still are only able to extract a subset of the total number of bugs (typically only 40%- 60% of those reported in the bug tracking system). Since the quality of change log comments varies across projects, it is possible that the output of the classification algorithm will include false positive and false negatives. Recent research by Bachmann et al. focusing on the Apache system is starting to shed light on the size of this missing data [21]. The impact of this data has been explored by Bird et al. who find that in the presence of missing data, the Bug Cache prediction technique [89] is biased towards finding less severe bug types [27].

Bug introducing data is incomplete.

The SZZ algorithm used to identify bug-introducing changes has limitations: it cannot find bug introducing changes for bug fixes that only involve deletion of source code. It also cannot identify bug-introducing changes caused by a change made to a file different

from the one being analyzed. It is also possible to miss bug-introducing changes when a file changes its name, since these are not tracked.

Selected classifiers might not be optimal.

We explored many other classifiers, and found that Naïve Bayes and SVM consistently returned the best results. Other popular classifiers include decision trees (e.g. J48), and JRIP. The average buggy F-measure for the projects surveyed in this chapter using J48 and JRip using feature selection was 51% and 48% respectively. Though reasonable results, they are not as strong as those for Naïve Bayes and SVM, the focus of this chapter.

Feature Selection might remove features which become important in the future.

Feature selection was employed in this chapter to remove features in order to optimize performance and scalability. The feature selection techniques used ensured that less important features were removed. Better performance did result from the reduction. Nevertheless, it might turn out that in future revisions, previously removed features become important and their absence might lower prediction quality.

5.10 Conclusion

This chapter has explored the use of feature selection techniques to predict software bugs. An important pragmatic result is that feature selection can be performed in increments of half of all remaining features, allowing it to proceed quickly. Between

3.12% and 25% of the total feature set yielded optimal classification results. The reduced feature set permits better and faster bug predictions.

The feature selection process presented in section 5.5 was empirically applied to eleven software projects. The process is fast performing and can be applied to predicting bugs on other projects. The most important results stemming from using the feature selection process are found in Table 5.4, which present F-measure optimized results for the Naïve Bayes classifier. A useful pragmatic result is that feature selection can be performed in increments of half of all remaining features, allowing it to proceed quickly. The average buggy is precision is quite high at 0.97, with a reasonable recall of 0.70. This result outperforms similar classifier based bug prediction techniques and the results pave the way for practical adoption of classifier based bug prediction.

From the perspective of a developer receiving bug predictions on their work, these figures mean that if the classifier says a code change has a bug, it is almost always right. The recall figures mean that on average 30% of all bugs will not be detected by the bug predictor. This is likely a fundamental limitation of history-based bug prediction, as there might be new types of bugs that have not yet been incorporated into the training data. We believe this represents a reasonable tradeoff, since increasing recall would come at the expense of more false bug predictions, not to mention a decrease in the aggregate buggy F-measure figure. Such predictions can waste developer time and reduce their confidence in the system.

In the future, when software developers have advanced bug prediction technology integrated into their software development environment, the use of the Bug

Prognosticator, classifiers with feature selection, will permit fast, precise, and accurate bug predictions. The age of bug-predicting imps will have arrived.

The next chapter introduces the Fix Suggester, a method to predict partial content of a bug fix.

Algorithm 1 Feature selection process for one project

1. Start with all features, F

2. For feature selection technique, f, in Gain Ratio, Chi-Squared, Significance Evaluation, Relief-F, Wrapper method using SVM, Wrapper method using Naïve Bayes, perform steps 3-6 below.

3. Compute feature Evaluations for using f over F, and select the top 50% of features with the best performance, $F/2$

4. Selected features, $selF = F/2$

5. While $|selF| \geq 1$ feature, perform steps (a)-(d)

   (a) Compute and store buggy and clean precision, recall, accuracy, F-measure, and ROC AUC using a machine learning classifier (e.g., Naïve Bayes or SVM), using 10-fold cross validation. Record result in a tuple list.

   (b) If f is a wrapper method, recompute feature scores over $selF$.

   (c) Identify $removeF$, the 50% of features of $selF$ with the lowest feature evaluation. These are the least useful features in this iteration.

   (d) $selF = selF - removeF$

6. Determine the best F-measure result recorded in step 5.a. The percentage of features that yields the best result is optimal for the given metric.

Table 5.4: Naïve Bayes (with Significance Attribute Evaluation) on the optimized feature set (binary)

Project Name	Features	Percentage of Features	Accuracy	Buggy Precision	Buggy Recall	Buggy F-measure	Clean Precision	Clean Recall	Clean F-measure	Buggy ROC
Apache	1098	6.25	0.93	0.99	0.63	0.77	0.92	1.00	0.96	0.86
Columba	1088	6.25	0.89	1.00	0.62	0.77	0.86	1.00	0.93	0.82
Eclipse	505	3.12	0.98	1.00	0.78	0.87	0.98	1.00	0.99	0.94
Gaim	1160	12.50	0.87	1.00	0.66	0.79	0.83	1.00	0.91	0.83
Gforge	2248	25	0.85	0.84	0.84	0.84	0.86	0.86	0.86	0.93
JCP	1310	3.12	0.96	1.00	0.80	0.89	0.95	1.00	0.97	0.90
Jedit	867	6.25	0.90	1.00	0.73	0.85	0.86	1.00	0.93	0.87
Mozilla	852	6.24	0.94	1.00	0.80	0.89	0.92	1.00	0.96	0.90
Plone	191	3.12	0.93	1.00	0.62	0.77	0.92	1.00	0.96	0.83
PSQL	2905	12.50	0.90	1.00	0.59	0.74	0.88	1.00	0.94	0.81
SVN	464	3.12	0.95	0.87	0.68	0.76	0.95	0.98	0.97	0.83
Average	1153.45	7.95	0.92	0.97	0.70	0.81	0.90	0.99	0.94	0.87

Table 5.5: SVM (with Gain Ratio) on the optimized feature set (binary)

Project Name	Features	Percentage of Features	Accuracy	Buggy Precision	Buggy Recall	Buggy F-measure	Clean Precision	Clean Recall	Clean F-measure	Buggy ROC
APACHE	549	3.12	0.89	0.87	0.51	0.65	0.90	0.98	0.94	0.75
COLUMBA	4352	25	0.79	0.62	0.71	0.66	0.87	0.82	0.84	0.76
ECLIPSE	252	1.56	0.96	1.00	0.61	0.76	0.96	1.00	0.98	0.81
GAIM	580	6.25	0.82	0.98	0.53	0.69	0.78	0.99	0.87	0.76
GFORGE	2248	25	0.84	0.96	0.67	0.79	0.78	0.98	0.87	0.82
JCP	1310	3.12	0.92	1.00	0.61	0.76	0.91	1.00	0.95	0.8
JEDIT	3469	25	0.79	0.74	0.67	0.70	0.81	0.86	0.83	0.76
MOZILLA	426	3.12	0.87	1.00	0.58	0.73	0.85	1.00	0.92	0.79
PLONE	191	3.12	0.89	0.98	0.44	0.60	0.88	1.00	0.93	0.72
PSQL	2905	12.50	0.85	0.92	0.40	0.56	0.84	0.99	0.91	0.7
SVN	464	3.12	0.92	0.80	0.56	0.66	0.94	0.98	0.96	0.77
Average	1522.36	10.08	0.87	0.90	0.57	0.69	0.87	0.96	0.91	0.77

Table 5.6: Naïve Bayes with 1% of all Features

Project Name	Features	Accuracy	Buggy Precision	Buggy Recall
Apache	175	0.90	1.00	0.46
Columba	174	0.82	1.00	0.38
Eclipse	161	0.97	1.00	0.69
Gforge	89	0.46	0.46	1.00
JCP	419	0.92	1.00	0.61
Jedit	138	0.81	1.00	0.48
Mozilla	136	0.88	1.00	0.60
Plone	61	0.89	1.00	0.45
PSQL	232	0.84	1.00	0.34
SVN	148	0.93	1.00	0.43
Gaim	92	0.79	1.00	0.45
Average	165.91	0.84	0.95	0.54

Table 5.7: Top 100 Bug Predicting Features

Project Name	Added Delta	Deleted Delta	New Revision Source	Changelog Features	Filename Features
Apache	49	43	0	0	5
Columba	26	18	50	4	2
Eclipse	34	40	26	0	0
Gaim	55	40	3	2	0
Gforge	23	30	40	6	1
JCP	39	25	36	0	0
Jedit	50	29	17	3	1
Mozilla	67	9	14	10	0
Plone	50	23	19	6	2
PSQL	44	34	22	0	0
SVN	48	42	1	9	0
Average	44.09	30.27	20.72	3.63	1

Table 5.8: Temporal Validation of the data sets of Table 5.4

Project Name	Features	Percentage of Features	Buggy F-measure	Traditional 10-fold Buggy F-measure
Apache	1098	6.25	0.63	0.77
Columba	1088	6.25	0.77	0.77
Eclipse	505	3.12	0.89	0.87
Gaim	1160	12.50	0.76	0.79
Gforge	2248	25	0.71	0.84
JCP	1310	3.12	0.82	0.89
Jedit	867	6.25	0.81	0.85
Mozilla	852	6.24	0.81	0.89
Plone	191	3.12	0.82	0.77
PSQL	2905	12.50	0.71	0.74
SVN	464	3.12	0.73	0.76
Average	1153.45	7.95	0.77	0.81

Table 5.9: Alternative F-measures on the data sets of Table 5.4

Project Name	Features	Percentage of Features	Buggy Precision	Buggy Recall	Buggy F-measure	Buggy $F_{0.5}$	Buggy F_2
APACHE	1098	6.25	0.99	0.63	0.77	0.89	0.68
COLUMBA	1088	6.25	1	0.62	0.77	0.89	0.67
ECLIPSE	505	3.12	1	0.78	0.87	0.95	0.82
GAIM	1160	12.50	1	0.66	0.79	0.91	0.71
GFORGE	2248	25	0.84	0.84	0.84	0.84	0.84
JCP	1310	3.12	1	0.80	0.89	0.95	0.83
JEDIT	867	6.25	1	0.73	0.85	0.93	0.77
MOZILLA	852	6.24	1	0.80	0.89	0.95	0.83
PLONE	191	3.12	1	0.62	0.77	0.89	0.67
PSQL	2905	12.50	1	0.59	0.74	0.88	0.64
SVN	464	3.12	0.87	0.68	0.76	0.82	0.71
Average	1153.45	7.95	0.97	0.70	0.81	0.90	0.74

Table 5.10: Comparisons of Our Approach to Related Work

Authors	Project(s)	Accuracy	Buggy F-measure	Buggy Precision	Buggy Recall	Granularity
Hata et al[a][73]	Eclipse Plugins	**-**	**71**	64	80	File
Shivaji et al.	Eclipse	**98**	**87**	100	78	Code Change
Gyimothy et al.[69]	Mozilla	**73**	**70**	70	70	File
Shivaji et al.	Mozilla	**94**	**89**	100	80	Code Change
Aversano et al.[18]	Multiple	**-**	**59**[b]	59	59[b]	File
Menzies et al.[113]	Multiple	**-**	**74**[b]	70[b]	79	File
Kim et al.[87]	Multiple	**78**	**60**	60	60	Code Change
Shivaji et al.	Multiple	**92**	**81**	97	70	Code Change

[a] Result is from two best historical data points

[b] Result presented are the best project results.

Chapter 6

Bug Fix Suggester

6.1 Introduction

Recent years have witnessed an increase of research interest in bug prediction techniques. A promising type of bug prediction algorithm uses machine learning techniques to predict whether a specific project change, as committed to a configuration management system, is a buggy change. Examples of such techniques can be found in the last chapter (and in [145]), in addition to [87, 18, 73] with the best techniques achieving high precision (changes predicted as buggy are very likely to be buggy), and reasonable recall (while not all buggy changes can be detected, a usefully large fraction are).

A notable challenge with such bug prediction algorithms is the nature of their output. They provide just a simple "buggy/clean" prediction for each change made to a project. That is, they take a change that a developer has just committed and then

simply state that the change is buggy, with no further feedback. We suspect that such bug prediction output might be more acceptable to developers should they additionally receive useful suggestions on how to go about fixing the predicted bug.

Ideally, a developer would like reasons for a code change being predicted as buggy along with suggestions on how to fix a bug. This chapter proposes a methodology which uses project history to statistically predict the content of future bug fixes to a buggy code change. Predicting the full content of a bug fix in verbatim is an extremely difficult problem. Instead, the proposed solution predicts and suggests unordered programming language tokens which are likely to appear in a future bug fix. "Fix Content" in this dissertation refers to the unordered keywords of a bug fix.

A developer can use fix suggestions when constructing a bug fix. Table 6.1 contains a buggy change and a part of its fix from the Argouml project. The bug, issue #1104, was reported as "UMLTextField generates propertyset events when handling one". The propertyset() method on a UMLTextField calls update(). Update() in turn sets a property, calling propertyset() again. An event storm results.

The fix suggestion approach in this chapter, called "Fix Suggester", predicted that the first code change is indeed buggy. It also predicted that the bug fix will involve the programming tokens "!getText().equals(.)", "getText().equals()", and "UMLTextProperty.setProperty(,,)". The actual bug fix was rather lengthy, but did modify "!getText().equals" as indicated in line 3 of the bug fix. A conditional added in a different intermediate commit is reflected in line 2 of the bug fix.

The goal of fix suggestions is to provide programming language tokens which

are likely to be present in an actual bug fix. Correctly predicting a fair amount of tokens present in the actual bug fix indicates a high level of precision. On the other hand, even if Fix Suggester's predicted tokens are in the actual fix, it might have missed many more tokens in the fix. It is desirable to have good coverage of tokens in the bug fix. This is also known as the recall of Fix Suggester. Having a reasonable degree of precision and recall will enable the system to be useful in practice.

We envisage a future where engineers have bug prediction and fix suggestions built in their development environment. They will receive a prediction on an impending code change followed by a list of potential bug fix suggestions. The presented approach starts with code changes from project history. It uses the change classification algorithm presented in chapter 3 on historical code changes. It is also possible to use a different algorithm that operates at the code change level. The set of fix changes for a particular buggy code change are used as input. Reasons for the bug are then given. These are typically keywords involved in a code change. Regions of the code change likely to be addressed by a bug fix are separately computed using the Bug Fix matrix, section 6.4.2.1. These are distinct from the reasons for a bug.

The fix prediction process learns from code changes extracted from a project's source control history. After sufficient training from history, a bug prediction classifier (possibly using the Bug Prognosticator of chapter 5) and a Bug Fix Matrix are built. The matrix contains a mapping of term frequencies from tokens in buggy changes to tokens in fix changes. Fix Suggestions for a new buggy code change are then predicted by jointly using the bug prediction classifier and the Fix Matrix.

6.2 Research Questions

This chapter explores the following research questions (all questions are mentioned in section 1.6).

RQ5. What is the prediction accuracy of the Fix Suggester on varying points of project history?

RQ5 addresses the accuracy and utility of the Fix Suggester in practice. Fix content is predicted at varying points of project history using only the information before that point in history. A classifier and a Bug Fix Matrix are used in combination to train and predict fix tokens that occur. Section 6.4.2 details the methods used for predicting bug fix content. Section 6.5.1 describes the results obtained from the approach. Validation is performed by comparing predictions against tokens present in actual bug fixes.

Once the fix suggestions are shown to be quite usable, the next logical question is which features are most important when predicting fix content. This leads to the next research question.

RQ6. What kind of features are ideal for generating Fix Suggestions?

The top hundred features input to the bug fix matrix by the classifier are investigated in further detail Section 6.5.2.

The primary contributions of this chapter is a novel method to predict bug fix content using inputs from a Bug Fix matrix and a classifier.

In the remainder of the chapter, we start by introducing the corpus of projects.

Next, the methodology proposed in the chapter are covered in section 6.4. This section starts by describing the overall workflow, additions to change classification, followed by the fix suggestion approach.

The stage is now set to discuss results. The chapter starts by detailing answers to the research questions described above. The chapter ends with a comparison to related work (Section 6.6), threats to validity (Section 7.5), and concluding remarks.

6.3 Corpus

The corpus of projects used in this chapter consist of ArgoUML[1], Eclipse JDT Core[2], jEdit[3], and Apache Lucene[4]. Instead of dealing with a small sample of revisions, we have chosen to go over long revision periods in each of these projects. The rationale was that open source project with long histories better resemble industrial projects. Table 6.2 contains details about the projects which were analyzed for this study.

6.4 Methodology

In this section, we distill the methodology used in the chapter, starting with the code change classifier followed by the Fix Suggester. An overview of the developer interaction workflow is depicted in Figure 6.1, repeated from the first chapter for convenience. The steps of the process are:

[1]http://argouml.tigris.org/
[2]http://www.eclipse.org/jdt/core/index.php
[3]http://www.jedit.org/
[4]http://lucene.apache.org/

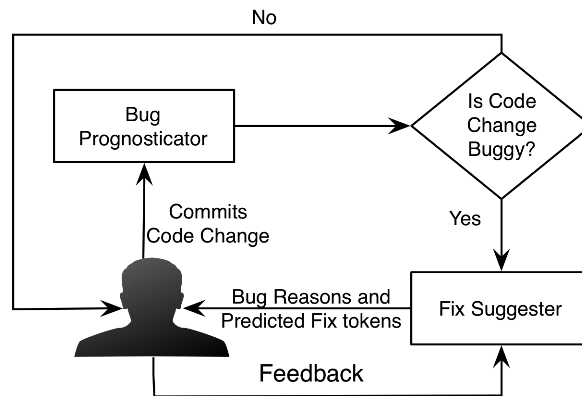


Figure 6.1: Developer Interaction Workflow

1. A code change is submitted.

2. A prediction is made on whether the entire change is buggy or clean using change classification. Change classification is explained in chapter 3.

3. If the code change is predicted to be buggy, suggest a partial code fix. The predicted fix tokens are presented to the user. The Fix content prediction algorithm is described in Section 6.4.2.

The next subsections describe these steps in detail.

6.4.1 Updates to Change Classification

Change classification is the algorithm used to label and predict code changes as buggy or clean. The workings of this algorithm were described in chapter 3. Updates to the algorithm are described in subsequent sections.

6.4.1.1 Feature Extraction

For this chapter, certain AST features were added to the typical change classification algorithm. The change distilling algorithm proposed by Fluri et al. [58] was used to extract the difference between abstract syntax trees (ASTs) built from the old and new revisions for each commit. The change distilling algorithm categorizes changes of ASTs into 48 types, according to the taxonomy defined by their previous work [57]. The number of occurrences for each change type was used as a feature.

A vocabulary of features was built from the first 500 revisions for each project in the corpus. This vocabulary of features was used by buggy, fix, and neutral revisions. Any new token brought in after revision 500 were not be used for bug prediction or Fix Suggestions. This was done In order to ensure that the Fix Suggester may be used after revision 500 without adding additional rows or columns to the Fix Matrix.

Each of these features can have largely differing sets of values. With differing value ranges, it is a challenging task to use the data. The next section deals with pre-processing features so that their value ranges are comparable.

6.4.1.2 Feature Pre-processing

This section explores the different kinds of features present in the data sets and techniques to align data for efficient analysis of code changes.

Keyword features can potentially have a large range of values if one uses their frequency of occurrence in code changes. There is a choice on how to interpret this feature, one can either record the presence of a keyword feature or record the number

of times the keyword appears in a code change.

For example, if a variable 'maxParameters' is present anywhere in the project, a binary interpretation of this feature records this variable as being present or absent, respectively 1 or 0 for a code change. A count interpretation would instead record the number of times it appears for a particular code change. It was observed that large values when using count tended to skew bug fix prediction. Additionally, employing a binary interpretation allows one to simplify fix prediction to the presence of a keyword in a code change, instead of the number of times a keyword will appear in a fix. Thus, a binary interpretation was used for keyword features. This limits the range of values to 0 and 1 for keyword features.

Numeric meta-data features such as LOC, cumulative change count, cumulative bug count, length of a change log can have unpredictable ranges as well. These were numerically scaled to the [0, 1] range using rudimentary numerical transformations [140].

AST features presented a different problem. While AST feature values do have a large range of values, the range is much smaller than keyword features. These were scaled to the [0, 1] range, similar to numeric meta-data features.

Categorical features such as author name, commit hour (0-23), and commit day (1-7) were not numerically scaled. One cannot conclude that a code change committed at 2pm is twice as important as a 1pm code change. These features were thus converted to binary terms. For example if commit day is 2, commit_day_is_2 would be 1 and commit_day_is_1, commit_day_is_3, .., commit_day_is_7 will all have zero values. The benefit of using binary values for these features is removing any numerical bias during

analysis.

The data is now transformed to the [0, 1] range.

Once the features are extracted, processed, and training is performed on sufficient revisions, a new code change can be classified as buggy or clean. The next section introduces an approach to predict bug fix content for a predicted buggy change.

6.4.2 Fix Suggester

The goal of the Fix Suggester is to accurately predict code tokens of a fix to the current buggy change. Predicting code tokens of a fix typically means predicting them in the order that they would appear in a fix. However, predicting the code tokens of a fix in verbatim is a very difficult problem. Fix Suggester, in this chapter, instead predicts unordered programming language tokens of a future bug fix to a presented buggy code change. In this chapter, contents of a bug fix are synonymous with the terms of a bug fix. These are in turn synonymous with fix suggestions.

We denote a buggy revision as B_n. The corresponding fix to this revision F_n occurs later in time.

The inputs to fix prediction are:

- A code change predicted to be buggy, revision B_n.
- All previous revisions annotated as buggy, a bug fix, or neither.

The output is a set of code tokens that are likely to be present in revision F_n. A good fix prediction will have a high intersection among predicted tokens and the

actual tokens of F_n.

A function that links tokens of B_n to the tokens of F_n is desirable for predicting fix content accurately. As most of the features extracted are keyword features, a link between buggy and fix keywords is helpful. This work focuses on statistical approaches to link buggy and fix tokens. The next section details the Bug Fix Matrix, an approach that predicts bug fix content using the modified data.

6.4.2.1 Bug Fix Matrix

The Bug Fix matrix generates a set of term frequencies among past buggy and fix tokens. It can then be used to predict tokens of a new buggy revision.

Suppose we had the following three keywords in a few code changes: File, File.open(), File.close(). An example fix matrix is depicted in Table 6.3. The matrix indicates that if there is a "File" keyword in the bug, among all the bug fixes, "File.close()" was present once in the bug fix. The bug fix in 2 cases added/modified "File" and in one case added a "File.close" when "File" is in the bug inducing change. When "File.open()" was present in the bug, the bug fix added a "File.close()". The matrix reflects a bug when a developer opens a file but forgets to close it. The bug fix calls File.close().

The Fix matrix generates a buggy to fix frequency map between all terms of the vocabulary generated after section 6.4.1.2. Feature i represents a particular token of the vocabulary. The left column of the matrix represents a token from a buggy code change. Each row of the matrix indicates the number of times a token shows up on the

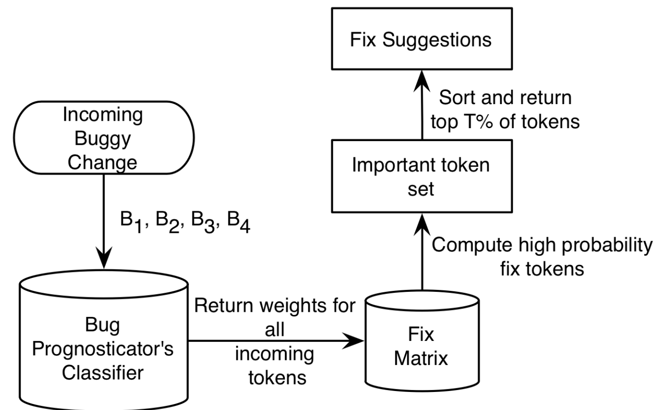


Figure 6.2: Fix Suggester

bug fix given presence of a token in the buggy change. For example, $M_{i,j} = 1$ means that for all buggy changes where token i was present, in only one of them token j was present in the bug fix. The Fix matrix can mention how often the same token i occurred in both the buggy change and the fix via $M_{i,i}$. In order to make sense out of the Fix Matrix's output, all term frequencies were scaled to the [0, 1] range. This avoids bias against frequently occurring tokens and simplifies the fix prediction process.

The Bug Fix Matrix can be used to predict fix content for an incoming buggy change. The Fix Suggester's algorithm is illustrated in Figure 6.2 and detailed in Algorithm 2.

In step 2, the fix matrix entries are adjusted using information from the code change classifier. The SVM algorithm assigns a weight to each feature, with a higher weight indicating that this feature is more useful than others when separating buggy changes from clean changes. We assume that these same features are also more useful

Algorithm 2 Fix Content Prediction Algorithm

1. A potentially buggy code change is given as input.

2. Create a fix matrix on the training set of bug inducing changes and their respective fixes up to this point.

3. Calculate the probability of occurrence in the bug fix for every token in the vocabulary.

4. For a new bug inducing code change, use the bug prediction classifier to return weights for all incoming buggy tokens.

5. Multiply the term frequency of each predicted fix token by the classifier weight for the source buggy feature if it is present.

6. Store all candidate fix tokens in imp_set.

7. Sort the list of features in imp_set by their conditional probability.

8. Return top T percent of high probability fix tokens from imp_set.

for predicting which keywords will appear in bug fixes as well, and hence we wish to emphasize these keywords in the fix matrix. To accomplish this, we multiply each matrix entry associated with a token by its SVM weight (as found in the SVM primal weight vector w in the Liblinear implementation [12]).

For each buggy token entry on the left side of the matrix, all tokens in the columns followed by their weight are returned to imp_set. Once the conditional probability of being a fix token is computed for all tokens in the vocabulary, a good cutoff is needed for the final step of extracting high probability fix tokens from imp_set. The top 10 percent of tokens with the highest conditional probabilities were returned. Fix Suggester's results are summarized in section 6.5.1.

An important practical issue with the Fix Matrix is the huge size it can encompass. Even using data from the first 500 revisions, the fix matrix vocabulary can still consist of about 50 thousand features. Having a 50 thousand by 50 thousand matrix can contain 2.5 billion entries. In order to make the algorithm perform well for this research, only the top 10% of tokens returned by the classifier were entered into the Fix Matrix. This is a radical compromise to simplify the problem. In order to increase recall at the cost of precision, it might be useful to include more features and investigate scalable solutions for large matrices.

A lesser practical issue is before predicting a bug fix for a buggy code change, one may not know a priori if a particular code change is buggy to start with. The SZZ algorithm for example, traces backward from a bug fix to a buggy change. Not knowing if a change is buggy until a bug fix is performed would mean that fix content prediction

itself could be less useful in practice.

However, the benefit of using a classifier driven approach is one can leverage previous work to make a buggy/clean prediction on a code change and return fix content only on those code changes which the classifier thinks are buggy. The code change classifier performed well on all corpus projects, when training on a portion of past project history to predict if future code changes are buggy or clean. A separate classifier was used for each project. It is possible to use an entirely different bug prediction algorithm to predict if a code change is buggy before applying fix prediction.

The next section details results from the Fix Suggester.

6.5 Results

6.5.1 Fix Suggester

The Fix Suggester was evaluated by training on p% of project history and testing on (100-p)% of project histories for p ranging from about 5 percent to 100 percent in 1 percent increments. As the vocabulary was generated using the first 500 revisions, no fix suggestions were presented when there are less than 500 revisions. In order to grade the efficiency of the Fix Suggester independent of the classifier, actual bug fixes from test data were compared to the Fix Suggester's fix tokens. In other words, performance of the Fix Suggester was graded on actual future bug fixes by passing in the buggy change and all history prior to the buggy change. Prior history was cleansed to ensure that no information from the future was visible. For example, if a historical

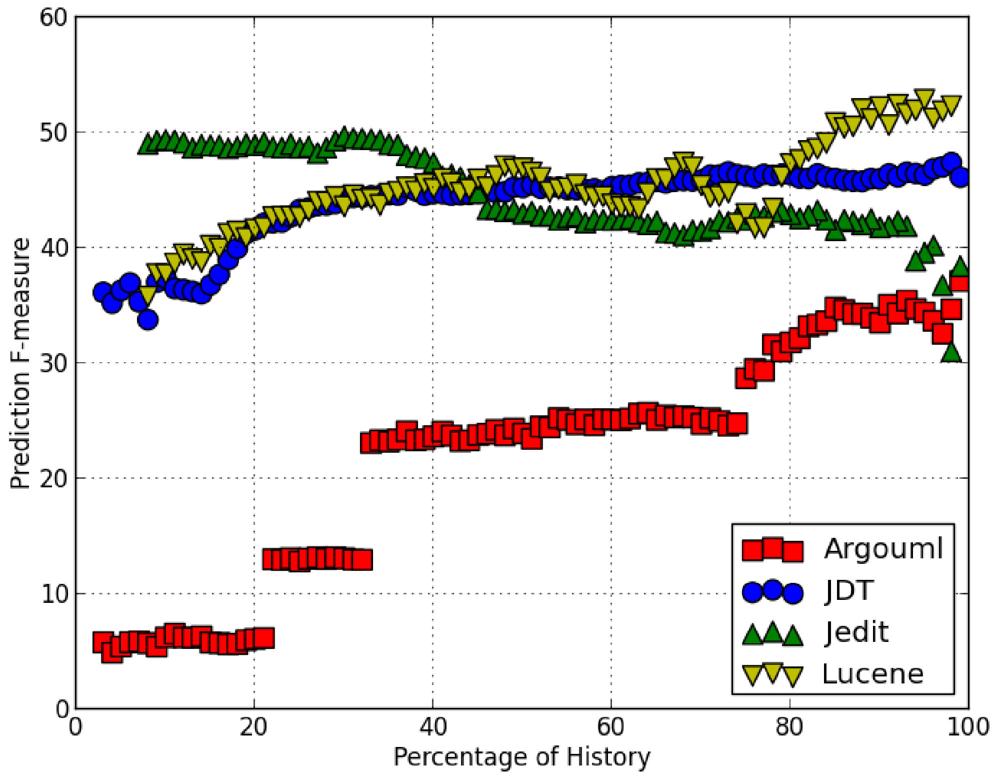


Figure 6.3: Fix Suggester F-measure on Project History

code change was labeled buggy due to a future fix, it is modified to reflect a neutral change in order to ensure that predictions do not unfairly use data from the future.

The Fix Suggester was able to achieve an average precision of 46.9% and a recall of 38.9% when averaging the evaluation points over all projects in the corpus. Table 6.4 contains averaged results by project. Figure 6.3 depicts a historical graph of predicted fix content F-measure versus ratio of project history. Unsurprisingly, fix prediction F-measure improves with more historical project data. It is encouraging that

after 500 revisions of project history, fix suggestions are usable for all projects except Argouml.

Overall, predicting bug fix content is a hard problem. Being able to correctly predict close to 47% of the actual content of bug fixes while covering almost 39% of all bug fix tokens is not a bad start. As precision can be improved at the cost of recall, and vice-versa, Figure 6.3 displays F-measure results instead. If a developer desires more precision at the cost of recall, it is possible to go beyond 47%.

A logical question that follows is what are the top bug reasons input to the Fix Matrix? A software developer might ask which code attributes are the most effective predictors of bugs. The next section deals with an analysis of the top 100 bug inducing features of each project.

6.5.2 Breakdown of Top 100 Bug Inducing Features

The top 100 most bug inducing features after training on 50% of project history are summarized in Table 6.5. The top 3 types are purely keyword related. Adding a certain keyword has the highest chance of creating a bug, followed by deletion, and introducing entirely new keywords to a file. New keyword features refer to the addition of keywords not seen before in a file. Meta-data and Changelog features are the next most prominent set of features.

Given that a small number of meta-data and changelog features are present in the projects to start with, a breakdown of the information conveyed by these features for each project is presented in Table 6.6. The name of the author seems to be an

important predictor in all projects except jEdit. For Argouml, 10pm is a bad time to commit code as is committing on a Wednesday. For Eclipse JDT, Thursday is not a good day for submitting code. The benefit of leveraging meta-data and changelog features is the potential to use this information across projects.

It is somewhat surprising that AST features do not make the top 100 feature list. While AST features were useful for fix prediction, they did not feature in the top 100.

6.5.3 Algorithm Time Analysis

The Fix matrix can be arbitrarily large as it encompasses a mapping from every term to every other term in the vocabulary. The vocabulary is built after the first 500 revisions. In order to reduce runtime, only the top 1% of terms were updated and returned. It took roughly 1 minute and 15 seconds to build the Fix Matrix (with a 2.26 Ghz Intel Core 2 Duo CPU and 8GB of RAM). After a code change is classified as bug inducing, compute fix terms using the Fix Matrix takes about 5-10 seconds (with a 2.26 Ghz Intel Core 2 Duo CPU and 8GB of RAM).

While one can build a Fix Matrix for every revision to suggest fixes, it is also possible to reuse a Fix Matrix for say the next 50 revisions. The performance of the Fix Suggester did not significantly degrade when this was attempted.

6.6 Comparison to Related Work

Suggestions for Bug Fixes can come from different techniques. The most common is via static analysis. Related work not using static analysis is also discussed.

6.6.1 Static Analysis Techniques

Predicting bug fix content is a challenging problem especially when using statistical techniques. The static analysis community has spent considerable effort in exploiting language semantics to suggest fixes to bugs. Popular tools using static analysis for fix suggestions include Findbugs [19], PMD [138], BLAST [121], FxCop [155] amongst many others. There are also approaches from literature which do not yet have downloadable tools available.

The method suggested in this chapter approaches the problem statistically. Comparing statistical analysis and static analysis was already mentioned in the first chapter but is repeated here for convenience. The general benefits of a statistical driven approach are:

- A focus on predicting fix suggestions which will actually be fixed in practice. Wedyan et al. has analyzed static analysis suggestions and found that less than 3% of the suggestions are actually fixed in practice [156]. When interacting with a few industrial settings, this number was found to be less than 0.5%. In contrast, the statistical driven approach presented has an average precision greater than 46% implying that almost half of the tokens returned by the Fix Suggester will

be used in future bug fixes.

- Project history is leveraged and automatically tailored for adaptive prediction of bug fixes relevant to the future code changes of a particular project. Historical trends can also be exploited. If a particular type of bug fix was popular at the onset of a project but diminished in significance soon, statistical fix content prediction will downplay the importance of that fix pattern.

There are advantages to static analysis when compared to statistical approaches including:

- The suggested bug fix is an exact solution which can often be proved in its effectiveness. In contrasts, a statistical approach is a probabilistic statement on the likely contents of a bug fix.

- The suggested fix and explanations can be well understood by humans.

6.6.2 Fix Content Prediction without Static Analysis

Kim et al.'s Bugmem provides fix suggestions using past revision history [90]. Bug and fix pairs are extracted from history. If an impending code change is similar to a previous bug, the prior buggy change and bug fix are displayed, and the developer is warned. This is a useful tool especially for developers who are new to a code base. They can be alerted of mistakes from project history.

Holmes and Murphy proposed an approach to extract structural components from example code and use them to assist coding when developers are working on similar

code [75].

6.7 Threats to Validity

Systems examined might not be representative of typical projects.

Four systems with large histories were examined. In spite of this, it is still possible that we accidentally chose systems that have better (or worse) than average fix suggestion prediction accuracy. Since we intentionally chose systems that had some degree of linkage between change tracking systems and change log text (to determine fix inducing changes), there is a project selection bias.

Systems are open source.

This threat is detailed in section 5.9. In contrast to chapter 5, all projects presented in this chapter are open source. This enlarges the relevance of the threat.

Bug fix and Bug Introducing data are incomplete.

This threat is detailed in section 5.9.

Selected classifiers might not be optimal.

We explored many other classifiers, and found that the SVM consistently returned the best results and has the most suitable infrastructure for the Fix Suggester. It is however possible that another classifier can do a better job.

6.8 Conclusion

This chapter introduces a novel statistical Fix Suggester approach that can predict unordered tokens of a bug fix. While predicting contents of a fix is a tough problem, the proposed approach is able to predict fix content with 46.9% precision and 38.9% recall on average. Section 6.5.1 shows that results are better than these average figures for many points in a project's history.

In the future, when software developers have advanced bug prediction technology integrated into their software development environment, the use of a Bug Fix matrix with a code change classifier will permit improved bug fix content prediction. With widespread use of integrated bug and fix content prediction, future software engineers can increase overall project quality by catching errors and deploying fixes in reduced time.

The next chapter raises the question of human feedback and how it can be leveraged to further enhance the Bug Prognosticator.

Table 6.1: Example bug and fix changes for Argouml Issue #1104

Bug-introducing Revision 754	Fix Revision 965
...	...
1: public void insertUpdate(final DocumentEvent p1){	1: public void insertUpdate(final DocumentEvent p1){
2: _textChanged=(_oldPropertyValue != null) &&	2: **if (_viaUserInput)** {
3: !getText().equals(_oldPropertyValue);	3: _textChanged=!getText().equals(_oldPropertyValue); }
4: handleEvent();	4: handleEvent();
5: }	5: }
...	...

Table 6.2: Projects

Name	**Revision Period**	**# of Total Commits**	**Buggy Commits**	**Fix Commits**	**Neutral Commits**
ArgoUML	01/26/1998 - 06/13/2011	17452	1516	536	15400
Eclipse JDT Core	06/05/2001 - 04/20/2011	17904	6640	4232	7032
jEdit	09/02/2001 - 07/02/2010	6050	3141	2046	863
Apache Lucene	09/11/2001 - 06/23/2011	5966	1923	1464	2579
Average	N/A	11843	3305	2069.5	6468.5
Average Percentage	N/A	100	27.9	17.5	54.6

Table 6.3: Bug Fix Matrix Example

	File	File.open()	File.close()
File	2	0	1
File.open()	0	0	1
File.close()	1	0	0

Table 6.4: Average Fix Content Prediction rate per project from Project History

Project	Precision	Recall	F-measure
Argouml	39.67	21.80	27.80
Eclipse JDT	45.26	43.81	44.47
jEdit	50.85	44.62	46.89
Lucene	51.93	45.18	48.28
Average	46.93	38.85	41.86

Table 6.5: Top 100 Bug Reasons

Project Name	Added Keywords	Deleted Keywords	New Source Keywords	Meta-data and Changelog Features
ArgoUML	72	0	1	27
Eclipse JDT	25	54	18	3
jEdit	94	0	1	5
Lucene	71	0	1	28
Average	65.5	13.5	5.25	15.75

Table 6.6: Meta-data and Changelog Features in the Top 100

Project	Features
ArgoUML	Author name, Changelog Messages and length, Commit hour - 10pm, Commit day - Wednesday, Cumulative bug count
Eclipse JDT	Author name, Commit day - Thursday, File path
jEdit	Changelog messages
Lucene	Author name, # of files copied, Commit hour - 9pm, Changed LOC, Cumulative bug count, Changelog messages

Chapter 7

Human Feedback

7.1 Introduction

While the results of fix content prediction in chapter 6 are a good start in the tough area of fix content prediction, one has not yet engaged a key resource, the users of the system. Both Bug and Fix predictions are ultimately going to be acted on or discarded by humans. Too many incorrect predictions will discourage humans from using the system in the future. It is vital to both engage humans in the defect prediction and the fix suggestion process.

In many other fields, combining the power of man and machine has yielded much fruit. Many internet firms use human feedback in various capacities to improve their product, e.g. Google, Yahoo, Microsoft, Ebay, and Amazon. In the game of chess, even an average chess player when playing in combination with a computer can outplay a computer playing alone. This has lead to increased popularity in advanced chess

tournaments where centaurs, man and machine combined, compete against each other for prizes [84]. Gary Kasparov, a former world chess champion, has drawn a parallel to the Moravec's paradox by showing that human and computer skills complement each other in the game of chess. He stated that in positions where computers are good, humans are weak, and vice versa [84]. Moravec's paradox states that simple human intuition is hard for machines to emulate [119]. However, complex human reasoning is typically easier for machines to compute. Thus, humans and machines can complement their skills effectively when solving problems.

Going back to defect prediction, the goal is to use human insights to guide bug predictions and fixes, while leveraging statistical machine power to guide human intuition. The next section deals with using human feedback to understand and improve bug prediction results.

7.2 Human Feedback on Fix Prediction

Before attempting human feedback, one has to consider practical constraints. Engineers have hectic day jobs and very little time to spare. When interacting with engineers from the industry on fix content feedback, it was quite difficult to get time and attention. Given that human time is expensive, they should only be engaged on the most critical code changes. Finally, making the feedback intriguing to humans is desirable. With the limited time available to provide feedback, lack of captivation can rapidly make matters worse.

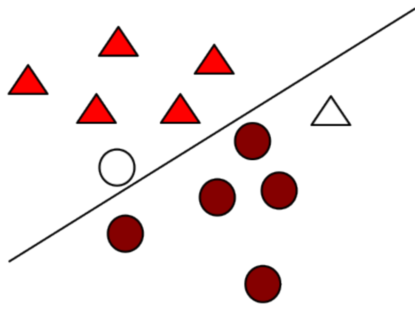


Figure 7.1: Example SVM Classifier in 2D

Algorithm 3 incorporates human feedback on bug prediction taking into account the above constraints.

The first two steps are self explanatory. Step 3 computes the bug predictors' confidence for each revision in the train set. Bug prediction at this step appears unrelated to fix content prediction. However, if the probability of a change being buggy is represented as $\mathrm{P}_{\mathrm{Buggy}}$, and the accuracy of fix content prediction is $\mathrm{Pr}_{\mathrm{Fix}}$, the overall probability of the fix content being valid is strongly influenced by $\mathrm{P}_{\mathrm{Buggy}}$. An extreme case is when $\mathrm{P}_{\mathrm{Buggy}}$ is zero, in which case predicting fix content for that change is meaningless. While multiplying $\mathrm{Pr}_{\mathrm{Fix}}$ by $\mathrm{P}_{\mathrm{Buggy}}$ to indicate the overall accuracy of fix content prediction does not necessarily make logical sense, $\mathrm{P}_{\mathrm{Buggy}}$ cannot be ignored.

One of the constraints with human feedback is that humans should be engaged for as short a duration as possible. While merely changing the label of a revision sounds simplistic, it is a low hanging fruit as the results in section 7.4 show.

For an SVM based bug prediction classifier, a hyperplane separating bugs

Algorithm 3 Human Feedback on Bug Prediction

1. A set amount of revisions from project history, e.g. p% is used as the train set.
2. The rest of project history is used as the test set, e.g. (100 - p)%.
3. Compute the prediction confidence score for each revision of the train set. The confidence score is how confident the SVM classifier is on correctly predicting a change as bug inducing or clean.
4. Mark the least confident incorrectly classified revision from the train set as a candidate for human feedback.
5. Consult with human on whether the candidate revision is buggy, a bug fix, neither, or both.
6. Apply algorithm 3 on the new data set.
7. Compare the precision and recall of fix prediction with and without human feedback.
8. Return a score to the human that is function of the delta in both precision and recall.
9. Return to step 3.

from clean changes is built. Figure 7.1 depicts a sample hyperplane in two dimensions. Typically an SVM is more confident of points further away from the hyperplane as opposed to those that are close. Points that are close to the boundary might easily be misclassified if the current hyperplane needs to be adjusted. In other words, points close to the hyperplane are most sensitive to a small change in the classifier. In figure 7.1, the hyperplane separates triangles from circles. The white circle and triangle are out of place. Both of those points are incorrectly classified and are in the wrong side of the separating hyperplane. This hyperplane is a line in 2D. If the triangles and circles are buggy and clean revisions, the top candidate for feedback would the revision represented by the white circle. This revision is picked over the white triangle as it is closer to the separating hyperplane.

Step 3 thus queries the point which the SVM is least confident on, and is incorrect. A question might arise on why the more confident incorrect revisions are not picked at this stage. The reason is to correct revisions one at a time, while improving classifier confidence gradually, starting from the least confident and proceeding to regions of greater confidence if the human is willing.

Step 4 proceeds to present this point to humans for feedback. This is a greedy algorithm as the model is recomputed based on a single point of feedback. Tong and Koller [153] show that myopic solutions are commonly used and work well in text classification. In order to relate this problem to the active learning context, every buggy/clean label that was derived from SZZ is treated as unlabeled as these points are machine labeled but not commented on by a human. In step four, the point most likely to have

incorrectly derived label is presented to humans for review.

Non greedy algorithms were also tested in this step and found to perform worse. An example of a non greedy algorithm is to present the N least confident revisions for human feedback. Empirically, it was found that presenting five or more least confident revisions to the user for human feedback was slower than presenting revisions one at a time.

The reason for this can be explained with an illustrative example. Say there are three revisions, R_1, R_2, and R_3. R_1 and R_2 are buggy, and R_3 is clean. If the classifier is given initial input that all revisions are clean, and it's confident that not all revisions should be clean, it can ask for human feedback. A human can give feedback one revision at a time or multiple revisions at a time. If the classifier deems that it is least confident on R_1, it can request feedback on it. Once a human labels that revision as buggy, the classifier might be able to deduce that revision 2 is buggy as well, and complete the cycle of feedback. In contrast, if the classifier returns the top two revisions that it is less confident about, i.e. revision 1 and 2, a human would have to waste time evaluating revision 2. The greedy algorithm gathers and applies feedback one revision at a time. It has the benefit that less overall feedback has to be ultimately delivered by a human.

Humans have a choice to label the presented revision as buggy, a bug fix, both, or neither. Several code changes are both a buggy and a fix when using SZZ. Human participants were allowed to look at the fix of the bug and the code change that it fixed in the case of a bug fix. Human participants were only allowed to look at revisions in

the train set. It is important to keep the test set totally hidden to get unbiased results. The correct judgment of a code change was left entirely up to human discretion.

A score is returned to the human after every feedback. This serves a dual purpose, to track the delta over a purely machine driven fix prediction algorithm and to provide more motivation for humans to deliver feedback. The computed score is a function of the precision and recall improved over the machine driven fix prediction approach of chapter 6. The exact formula is

$$score = M * (Pr_{FPH} - Pr_{FPM} + R_{FPH} - R_{FPM})$$

where Pr_{FPH} is the precision of fix prediction when given human feedback and Pr_{FPM} is the precision of fix prediction for the machine driven approach. Correspondingly, R_{FPH} and R_{FPM} are the recall of the fix prediction process when driven by humans and machines respectively. M is a multiplication scaling factor which transforms the score into larger numbers for better human appreciation. M was set to 10 in the human feedback experiments.

The next section details sample questions shown to users during the human feedback process, this is followed by a user study on the practical utility of the Fix Suggester.

7.2.1 Sample Questions Posed during the Human Feedback Process

A sample of actual questions with code hyperlinks are reproduced below. Clicking on a file will allow the user to inspect file contents.

Commit Log
Update for 0.20. git-svn-id:
http://argouml.tigris.org/svn/argouml/trunk@9675
a161b567-7d1e-0410-9ef9-912c70fedb3f

Repository Snapshot

Type File
M documentation/manual/argomanual.xml
M www/features.html
Do you think:
This is a Bug Fix? Yes/No
This is buggy? Yes/No

This commit is classified as buggy

Bug Fixing Commits
43000

Commit Log
Increased maximum heap git-svn-id:
https://jedit.svn.sourceforge.net/svnroot/jedit/jEdit/trunk@7626
6b1eeb88-9816-0410-afa2-b43733a0f04e

Repository Snapshot

Type File
M build.xml
M package-files/linux/jedit
M package-files/os2/jedit.cmd
M package-files/windows/jedit.bat
M package-files/windows/win32installer.iss

Do you think:
This is a Bug Fix? Yes/No
This is buggy? Yes/No

This commit is classified as buggy

Bug Fixing Commits
14569
14327

Commit Log
fixed compile error git-svn-id:
https://jedit.svn.sourceforge.net/svnroot/jedit/jEdit/trunk@4467
6b1eeb88-9816-0410-afa2-b43733a0f04e

Repository Snapshot

Type File
M doc/CHANGES.txt
M doc/TODO.txt
M modes/prolog.xml
M modes/python.xml
M org/gjt/sp/jedit/Buffer.java
M org/gjt/sp/jedit/EBComponent.java
M org/gjt/sp/jedit/EBMessage.java
M org/gjt/sp/jedit/EBPlugin.java
M org/gjt/sp/jedit/EditAction.java
M org/gjt/sp/jedit/EditBus.java

Previous 1 2 3 4 Next

Do you think:
This is a Bug Fix? Yes/No
This is buggy? Yes/No

Bug Introducing Commits
12666
12633
12771
12463
12667
12475
12602
12822
12712
12806
12366

```
12344
12861
12190
12229
12328
12592
12587
12851
12422
12818
12862
12626
12362
12241
12566
12423
12456
12385
12231
This commit is classified as buggy

Bug Fixing Commits
14414
```

7.3 Fix Suggester User Study

The results of fix content prediction in chapter 6 seem quite promising. It would seem that there is not much to ask beyond a good level of precision and recall. In practice, however, having a precision and recall of less than 50% each might confuse humans when working on a bug fix. In addition, the utility of the proposed keywords should be better understood. The final benefit of a user study focusing on industrial software engineers is evaluating the suitability of corporate adoption of the

Fix Suggester. The next section recaps the relevant research questions for convenience.

7.3.1 Research Questions

RQ8 When engineers inspect the bug fix change log, do they find that the Fix Suggester's keywords are relevant to the actual bug fix?

RQ9 Does reviewing the Fix Suggester's keywords influence the investigation for the bug fix?

The next section explains the user study process.

7.3.2 User Study Procedure

Fix Suggester results were shown to 19 users for both RQ8 and RQ9. To make analysis convenient, the respondents were asked to complete an online survey showing partial code diff data. SurveyMonkey was used for online delivery of surveys The majority of participants were industrial software engineers. All participants had some knowledge of Java.

In the case of RQ8, users were asked to indicate if the suggested keywords intersected with the actual bug fix using a Likert scale of 3 options. Participants were shown the Bug Fix change log, parts of the actual bug fix, and the Fix Suggester's keywords.

Not Helpful The predicted keywords did not intersect at all with the actual bug fix.

Somewhat Helpful The predicted keywords partially intersect with the actual bug fix.

Helpful The predicted keywords intersect with key portions of the bug fix.

A response of “Not helpful” indicates that despite statistical intersections, the predicted keywords did not actually reflect in the presented extracts of the bug fix. Somewhat helpful connotes clear intersection but not with key portions of the bug fix. Helpful depicts that an intersection exists between the suggestions and key regions of the bug fix.

For RQ9, subjects were shown the bug fix change log comment. They were also shown parts of the actual buggy change. They were asked to form a set of candidate keywords for the bug fix. After viewing the suggested keywords, they were asked if their set of candidate keywords was altered after viewing the fix suggestions. This experiment replicates potential industrial usage of the Fix Suggester. During a typical business day, one can envisage that the content of the bug fix is not yet known, and the Fix Suggester's suggestions have to be evaluated on face value.

The bug fix change log comments were shown to the users in order to guide their search for the fix. It is hard to isolate potential bug fixes on large set of code changes without any direction. In practice, a bug report, or a customer experience with a bug can be substitutes for the change log comment of the actual bug fix.

The feedback options for RQ9 include

Not Helpful My candidate keyword list was unaffected

Somewhat Helpful My candidate keyword list changed somewhat due to the fix suggestions

Helpful My candidate keyword list was influenced by the fix suggestions

The option "Not helpful" in this case means an unaltered candidate keyword list. Somewhat helpful indicates that the candidate keyword list had a moderate change. Helpful means the candidate code list was strongly influenced by the suggestions. Users provided feedback on randomly selected code changes on every project for which fix suggestions were computed.

The next section displays some actual user study questions shown to users. This is followed by details of the improvement gained by human feedback on fix prediction. Finally, the results of the qualitative study on the Fix Suggester are presented.

7.3.3 User Study Questions

Actual user study text follows, starting with preparatory instructions followed by each question.

The instructions should be self-explanatory. The questions ask you to evaluate the Fix Suggester's keywords. In a industrial engineering environment, I want to assess the utility of having access to the Fix Suggester's keywords right in front of you before you work on a bug fix.

Using information for code change history, the Fix Suggester predicts keywords

and other properties of a fix.

Explanation for keywords

- new_source_* means that the * keyword is predicted as going to be added to a new file for the bug fix.
- added_delta_* means that * will be added to an existing file.
- deleted_delete_* means that * will be deleted from an existing file.

---

1. Looking at the bug fix changelog, can you state if the predicted keywords are relevant to the actual bug fix?

```
Predicted keywords:
new_source__i, new_source__id, new_source__if,
new_source__iff, new_source__lock, new_source__makelock,
new_source__implements, new_source__in,
new_source__index, new_source__indexreader, new_source__indirect

Fix Commit log: Decouple locking from Directory:
LUCENE-635 git-svn-id: http://svn.apache.org/repos/asf/lucene/java/
trunk@437897 13f79535-47bb-0310-9956-ffa450edef68

Sample Actual Bug Fix Text:
The plusses refer to added code and the minuses refer to removed code.
a/.. refers to the old file
and b/src.. refers to the new file.
The diff format is similar to a git diff.

--- a/src/java/org/apache/lucene/store/Directory.java
+++ b/src/java/org/apache/lucene/store/Directory.java
@@ -29,9 +29,18 @@ import java.io.IOException;
  *
 implementation of an index as a single file;
  *
```

```
  *
+ * Directory locking is implemented by an instance of {@link
+ * LockFactory}, and can be changed for each Directory
+ * instance using {@link #setLockFactory}.
+ *
  * @author Doug Cutting
  */
 public abstract class Directory {
+
+  /** Holds the LockFactory instance (implements locking for
+   * this Directory instance). */
+  protected LockFactory lockFactory;
+
   /** Returns an array of strings, one for each file
    in the directory. */
   public abstract String[] list()
        throws IOException;
@@ -75,9 +84,43 @@ public abstract class Directory {
   /** Construct a {@link Lock}.
    * @param name the name of the lock file
    */
-  public abstract Lock makeLock(String name);
+  public Lock makeLock(String name) {
+      return lockFactory.makeLock(name);
+  }

   /** Closes the store. */
   public abstract void close()
        throws IOException;
+
+  /**
+   * Set the LockFactory that this Directory instance should
+   * use for its locking implementation.  Each * instance of
+   * LockFactory should only be used for one directory (ie,
+   * do not share a single instance across multiple
+   * Directories).
+   *
+   * @param lockFactory instance of {@link LockFactory}.
+   */
+  public void setLockFactory(LockFactory lockFactory) {
+      this.lockFactory = lockFactory;
+      lockFactory.setLockPrefix(this.getLockID());
+  }
```

```
+  /**
+   * Get the LockFactory that this Directory instance is using for its
+     locking implementation.
+   */
+  public LockFactory getLockFactory() {
+      return this.lockFactory;
+  }
+
+  /**
+   * Return a string identifier that uniquely differentiates
+   * this Directory instance from other Directory instances.
+   * This ID should be the same if two Directory instances
+   * (even in different JVMs and/or on different machines)
+   * are considered "the same index".  This is how locking
+   * "scopes" to the right index.
+   */
+  public String getLockID() {
+      return this.toString();
+  }
 }
```

Not Helpful The predicted keywords did not intersect with the actual bug fix at all

Somewhat Helpful The predicted keywords partially intersect with the actual bug fix

Helpful The predicted keywords intersect with the key portions of the bug fix

Comments?

---

2. Looking at the bug fix changelog, can you state if the predicted keywords are relevant to the actual bug fix?

```
Predicted keywords:
new_source__i, new_source__id, new_source__if,
```

```
new_source__iff, new_source__implements, new_source__in,
new_source__cachingqueue, new_source__index, new_source__indexed,
new_source__indexreader, new_source__indexwriter

Fix Commit log: "LUCENE-1223: fix lazy field loading to not allow string
field to be loaded as binary, nor vice/versa git-svn-id:
http://svn.apache.org/repos/asf/lucene/java/trunk@636568
13f79535-47bb-0310-9956-ffa450edef68"

Sample modification from fix:
--- a/src/java/org/apache/lucene/index/FieldsReader.java
+++ b/src/java/org/apache/lucene/index/FieldsReader.java
@@ -235,15 +235,15 @@ final class FieldsReader {
   }

   private void addFieldLazy(Document doc, FieldInfo fi, boolean binary,
boolean compressed, boolean tokenize) throws IOException {
-    if (binary == true) {
+    if (binary) {
       int toRead = fieldsStream.readVInt();
       long pointer = fieldsStream.getFilePointer();
       if (compressed) {
         //was: doc.add(new Fieldable(fi.name, uncompress(b),
// Fieldable.Store.COMPRESS));
-        doc.add(new LazyField(fi.name, Field.Store.COMPRESS, toRead,
-        pointer));
+        doc.add(new LazyField(fi.name, Field.Store.COMPRESS, toRead,
+        pointer, binary));
       } else {
         //was: doc.add(new Fieldable(fi.name, b, Fieldable.Store.YES));
-        doc.add(new LazyField(fi.name, Field.Store.YES, toRead,
-        pointer));
+        doc.add(new LazyField(fi.name, Field.Store.YES, toRead,
+        pointer, binary));
       }
       //Need to move the pointer ahead by toRead positions
       fieldsStream.seek(pointer + toRead);
@@ -257,7 +257,7 @@ final class FieldsReader {
         store = Field.Store.COMPRESS;
         int toRead = fieldsStream.readVInt();
         long pointer = fieldsStream.getFilePointer();
-        f = new LazyField(fi.name, store, toRead, pointer);
+        f = new LazyField(fi.name, store, toRead, pointer, binary);
```

```
         //skip over the part that we aren't loading
         fieldsStream.seek(pointer + toRead);
         f.setOmitNorms(fi.omitNorms);
@@ -266,7 +266,7 @@ final class FieldsReader {
         long pointer = fieldsStream.getFilePointer();
         //Skip ahead of where we are by the length of what is stored
         fieldsStream.skipChars(length);
-        f = new LazyField(fi.name, store, index, termVector, length,
-        pointer);
+        f = new LazyField(fi.name, store, index, termVector, length,
+        pointer, binary);
         f.setOmitNorms(fi.omitNorms);
       }
       doc.add(f);
@@ -385,17 +385,19 @@ final class FieldsReader {
     private int toRead;
     private long pointer;

-    public LazyField(String name, Field.Store store, int toRead,
-    long pointer) {
+    public LazyField(String name, Field.Store store, int toRead,
+    long pointer, boolean isBinary) {
       super(name, store, Field.Index.NO, Field.TermVector.NO);
       this.toRead = toRead;
       this.pointer = pointer;
+      this.isBinary = isBinary;
       lazy = true;
     }

-    public LazyField(String name, Field.Store store, Field.Index index,
-     Field.TermVector termVector, int toRead, long pointer) {
+    public LazyField(String name, Field.Store store, Field.Index index,
+    Field.TermVector termVector, int toRead, long pointer,
+    boolean isBinary) {
       super(name, store, index, termVector);
       this.toRead = toRead;
       this.pointer = pointer;
+      this.isBinary = isBinary;
       lazy = true;
     }

@@ -413,25 +415,27 @@ final class FieldsReader {
      /* readerValue(), binaryValue(), and tokenStreamValue()
```

```
       must be set. */
     public byte[] binaryValue() {
       ensureOpen();
-      if (fieldsData == null) {
-        final byte[] b = new byte[toRead];
-        IndexInput localFieldsStream = getFieldStream();
-        //Throw this IO Exc
```

Not Helpful The predicted keywords did not intersect with the actual bug fix at all

Somewhat Helpful The predicted keywords partially intersect with the actual bug fix

Helpful The predicted keywords intersect with the key portions of the bug fix

Comments?

---

3. Given the changelog [comment] of the bug fix, mentally form a set of keywords needed for the bug fix. Does your set of keywords change after looking over the suggested keywords?

```
Fix Changelog: LUCENE-1959 Add MultiPassIndexSplitter. git-svn-id:
http://svn.apache.org/repos/asf/lucene/java/trunk@824798
13f79535-47bb-0310-9956-ffa450edef68

Predicted keywords:
new_source__its, new_source__contains, new_source__documents,
new_source__on, new_source__once, new_source__one, new_source__open,
new_source__copy, new_source__count, deleted_delta__new, new_source__id,
new_source__or, new_source__created, new_source__return,
new_source__if, new_source__org, new_source__current, new_source__other

Current "Buggy" Code change:
```

```
--- /dev/null
+++ b/contrib/misc/src/java/org/apache/lucene/index/IndexSplitter.java
@@ -0,0 +1,163 @@
+/**
+ * Licensed to the Apache Software Foundation (ASF) under one or more
+ * contributor license agreements.  See the NOTICE file distributed with
+ * this work for additional information regarding copyright ownership.
+ * The ASF licenses this file to You under the Apache License, Version 2.0
+ * (the "License"); you may not use this file except in compliance with
+ * the License.  You may obtain a copy of the License at
+ *
+ *     http://www.apache.org/licenses/LICENSE-2.0
+ *
+ * Unless required by applicable law or agreed to in writing, software
+ * distributed under the License is distributed on an "AS IS" BASIS,
+ * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
+ * See the License for the specific language governing permissions and
+ * limitations under the License.
+ */
+package org.apache.lucene.index;
+
+import java.io.File;
+import java.io.FileInputStream;
+import java.io.FileOutputStream;
+import java.io.IOException;
+import java.io.InputStream;
+import java.io.OutputStream;
+import java.text.DecimalFormat;
+import java.util.ArrayList;
+import java.util.List;
+
+import org.apache.lucene.store.FSDirectory;
+
+/**
+ * Command-line tool that enables listing segments in an
+ * index, copying specific segments to another index, and
+ * deleting segments from an index.
+ *
+ *
This tool does file-level copying of segments files.
+ * This means it's unable to split apart a single segment
+ * into multiple segments.  For example if your index is
+ * optimized, this tool won't help.  Also, it does basic
```

```
+ * file-level copying (using simple
+ * File{In,Out}putStream) so it will not work with non
+ * FSDirectory Directory impls.


+ *
+ *
NOTE: The tool is experimental and might change
+ * in incompatible ways in the next release.  You can easily
+ * accidentally remove segments from your index so be
+ * careful!
+ */
+public class IndexSplitter {
+  public SegmentInfos infos;
+
+  FSDirectory fsDir;
+
+  File dir;
+
+  /**
+   * @param args
+   */
+  public static void main(String[] args) throws Exception {
+    if (args.length < 2) {
+      System.err
+          .println("Usage: IndexSplitter  -l (list the segments
+           and their sizes)");
+      System.err.println("IndexSplitter   +");
+      System.err
+          .println("IndexSplitter  -d (delete the following segments)");
+      return;
+    }
+    File srcDir = new File(args[0]);
+    IndexSplitter is = new IndexSplitter(srcDir);
+    if (!srcDir.exists()) {
+      throw new Exception("srcdir:" + srcDir.getAbsolutePath()
+          + " doesn't exist");
+    }
+    if (args[1].equals("-l")) {
+      is.listSegments();
+    } else if (args[1].equals("-d")) {
+      List segs = new ArrayList();
+      for (int x = 2; x < args.length; x++) {
```

```
+          segs.add(args[x]);
+        }
+        is.remove((String[]) se
```

Not Helpful My candidate keyword list was unaffected

Somewhat Helpful My candidate keyword list changed somewhat due to the Fix Suggestions

Helpful My candidate keyword list was influenced by the Fix Suggestions

Comments?

7.4 Results

7.4.1 Human Feedback Improvement on Fix Prediction

Human feedback gave an average improvement of 13.2% precision and 8.8% recall over machine generated fix prediction results (of chapter 6) after feedback was given on only 10 revisions! This is a surprising result indicating that even minute feedback can be very beneficial. Table 7.1 depicts the improvement by project. Users were restricted to providing feedback on the first half of project history, with their feedback being applied on the second half of project history. Table 7.1 shows a gain in both precision and recall after ten revisions.

Successful human feedback mainly consisted of the following activities.

Table 7.1: Average Bug Prediction Results after Feedback on 10 revisions per project

Project	% Precision Gain	% Recall Gain
Argouml	40.75	23.55
Eclipse JDT	3.02	6.59
Jedit	4.65	4.70
Lucene	4.19	0.43
Average	13.15	8.82

Correcting Incorrectly Labeled Fixes

These are code changes which are labeled as fixes in the change log, but are not typical bug fixes. A few of these fixes included minor fixes to documentation, java docs, and configuration files. There were also bug fixes that were code refactorings. While these were labeled as fixes in the change log, human judgment indicated otherwise. Notably, the machine learner also diagnosed these revisions as likely to be incorrect and suggested during the feedback step that these were incorrectly labeled.

De-emphasizing Minor Bug Fixes

A typical minor bug fix observed in a projects is a bug fix that removes compilation warnings. Humans participating in the study judged the code change as not being a true bug fix. Minor bug fixes were incorrectly lowering the precision and recall of predicting bug fix content.

As human feedback was only used for 10 revisions, low hanging fruit might have been uncovered. It is possible that as feedback is given on many more revisions, the kind of feedback would be rather different. The next section details feedback as a case study.

7.4.1.1 Feedback Improvement Case Study

Feedback was conducted on ten humans, six of them are engineers working in the industry and four are student researchers or faculty. Humans were given a score for their feedback during the entirety of their participation. Each participant gave feedback on ten revisions per project, for two different projects. For some individual human evaluations, there were points of negative score gains in the interim, before ten revisions were complete. Interestingly, this motivated humans to provide more feedback and improve their score.

Overall, the industrial participants in particular were intrigued during the process of providing feedback which can be used to predict bug fix content. After the evaluation the participants discussed the possibility of adopting such a system in their daily professional lives. Their concerns are listed below.

Speed of applying human feedback

The method provided in the chapter can take up to two minutes to apply human feedback and request feedback on the next computed revision. This is due to the fact that a new SVM is built after human feedback on a particular revision. The bottleneck is computing the next revision for human feedback. In practice, this

can be solved in a variety of ways. The simplest is to process human feedback asynchronously. The feedback score and the next revision would be sent in an email or by a comparable asynchronous solution. There is also research on improving active learning speed with support vector machines [65]. Finally, using fast non SVM based online learners like Vowpal Wabbit can speed up active learning [97] . One of the participants indicated that providing feedback on incorrectly labeled fixes especially when the feedback took two minutes to process is boring. He mentioned that he would not mind providing feedback on incorrect fixes if the feedback turn around time was a matter of seconds.

Human Feedback score

When participants see an improved positive score, they get a feeling that their feedback was correct. Conversely, a negative score creates a negative feeling. In reality, both feelings can be incorrect. While accurate feedback would converge towards an increased delta on both precision and recall, it is well possible that during the short-term, negative scores can result from correct feedback. An example is when there are a few revisions the classifier is not confident on, say R_1, R_2, and R_3. The correct classification is a bug fix for all three, and currently all three are marked as buggy. If feedback was only provided on R_1, but not on the others, it is possible that a model with R_1 marked as a bug fix, but R_2 and R_3 marked as buggy will score worse than the initial machine model of all three marked as buggy. After feedback is provided on all three revisions, there can be a

marked improvement over machine based fix prediction. An idea worth exploring is displaying revisions that are most responsible for grading human feedback as negative.

Collaborative Feedback

Incorporating feedback from a few humans on a particular revision can improve the prediction process taking less time away from each human. Undoubtedly, poor feedback from one or two humans can have an adverse effect on the system. On the whole, given that human feedback on average substantially improved fix content prediction as shown in table 7.1, a collaborative feedback model appears promising.

The next section details results of the qualitative study performed on the Fix Suggester.

7.4.2 Fix Suggester Qualitative Study

RQ8 states "When engineers inspect the bug fix change log, do they find that the suggested keywords are relevant to the actual bug fix?" 69.5% of the surveyed engineers found the suggested keywords to be somewhat or quite helpful. 30.5% of engineers on the other hand found the suggested keywords to not be helpful, they did not see any intersection with the actual bug fix.

RQ9 states "Does reviewing the Fix Suggester's keywords influence the investigation for the bug fix?" 67.4% of engineers found the suggested keywords to be

somewhat or quite helpful. 32.6% of engineers found the suggested keywords to be not helpful, stating that their candidate keyword list was unaffected by the suggestions.

The distribution of utility of the suggestions for RQ1 and RQ2 is respectively shown in figures 7.2 and 7.3. There was not much deviation from one project to another. It is promising that engineers on average found the Fix Suggester to be somewhat useful for both RQ8 and RQ9.

The general feeling was that the Fix Suggester is quite useful for engineers new to a codebase. Typical co-occurrence patterns of a bug fix were exposed especially for user interface driven projects such as Jedit and Argouml. The patterns are not limited to simple co-occurrence. The Fix Suggester is perhaps a customized filtered version of a co-occurrence driven solution based on fixes applied to similar bugs. These patterns tend to help engineers new to a project. A practical example is the knowledge that getPreferredSize, actionPerformed, setLayout, propertiesChanged are keywords that should be included if bug fix involves changing a UI action in Jedit. Another is that a bug fix on the Lucene index will require changing documents, created, current (document), open (document), and (document) count.

However, there are a few areas for improvement. Firstly, about 30% of users finding the suggestions ineffective is a clear area to work on for both research questions. The more challenging issue is to sway users who found it to be 'Somewhat Helpful'. The following suggestions were based on the verbal feedback provided by the users.

Large Code Changes The Fix Suggester did not fare well on large code changes

for both RQ8 and RQ9. For RQ8, it was especially hard for engineers to look for intersections when code spanned several files. On RQ9, a large keyword list tended to confuse engineers. It might be interesting to adaptively trim the set of recommended keywords on a particular code change. It might also be possible to trim the suggestion list based on how tolerant the user is. Those users finding it not helpful would have benefited from a smaller set of suggestions.

Reducing Search Time Having extraneous keywords can confuse engineers in practice. For RQ9, this means the search for a bug fix can be derailed by poor/less likely keyword choices. Engineers did not want to waste time on a suggested keyword if it was unlikely to be needed for this fix. A model that tried to minimize wasted time at the cost of missing out a few relevant keywords appealed to them. At first, this appears to be a simple precision/recall tradeoff. However, it is actually a more complex request for lower recall but saving potentially significant human time. Human time can be saved by not including plausible but ineffective keywords. A plausible but incorrect keywords might lure unsuspecting engineers to waste time while considering viabilities for these types of keywords.

More Detail To sway engineers from viewing the suggestions as somewhat helpful, they seemed to desire more detail on why those keywords need to change in a bug fix. One idea would be to integrate the keyword suggestions with static analysis techniques before presenting them. Static analysis tools like Findbugs often report several detailed violations. Engineers typically ignore the bulk of these violations.

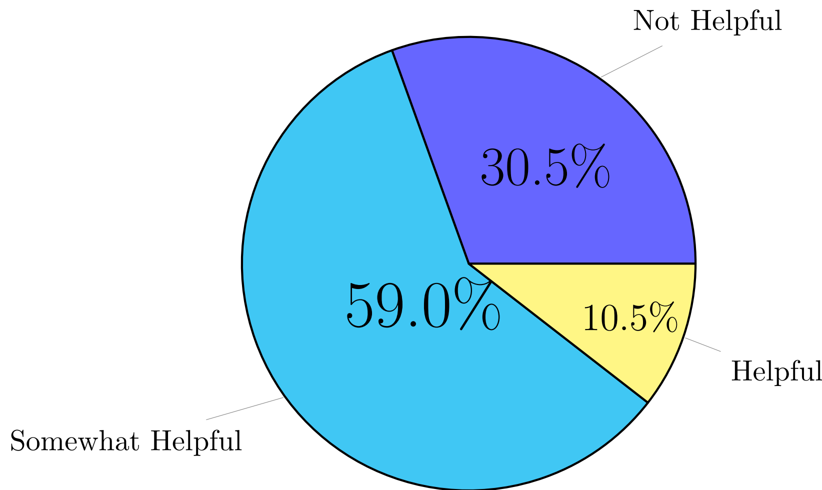


Figure 7.2: RQ8 Feedback

If the violations can be filtered by the Fix Suggester, they might be more likely to pay attention given that the Fix Suggester provides statistically relevant bug fix information. Statistics backed by the explanation power of static analysis can possibly be combined to make the suggestions effective.

User Personalization About 30% of users found the suggestions to be not helpful. It might be useful to target suggestions at these users adaptively. If a user was unhappy with a suggestion due to too many keywords, less keywords can be offered the next time. Personalizing the Fix Suggester at the user level will likely increase practical satisfaction.

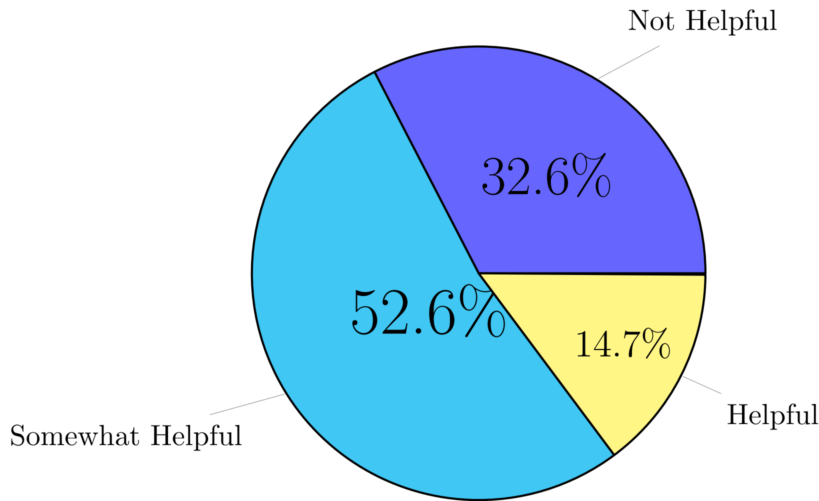


Figure 7.3: RQ9 Feedback

7.5 Threats to Validity

All threats mentioned in chapters 5 and 6 are applicable here. Independent threats include:

Humans consulted for feedback may not be representative. We selected six people from the industry, and four from academia for improving bug prediction. It is possible that these individuals provide better or worse feedback than the typical industrial or academic participant. On the qualitative study for fix prediction, almost all participants were from the industry. However, it is again possible that these individuals do not reflect typical industrial engineers.

Humans could have lead classifiers in the wrong direction. It is possible that incorrect feedback worsened the presented results. In practice, human feedback can potentially have negative effects and worsen the utility of the Fix

Suggester. We expect that extended human feedback should lead to an overall increase in performance.

Chapter 8

Future Work

8.1 Introduction

Chapters 5, 6, and 7 introduce improvements in bug prediction, fix content prediction, and human feedback respectively. This chapter discusses opportunities for future work arranged by the area of contribution.

8.2 Opportunities

Human Feedback Increase the permitted types of human feedback.

Human Feedback Provide automated mechanisms to validate human feedback.

Fix Suggester Refine the Fix Matrix to hold complex relationships between buggy and fix terms.

Bug Prognosticator Predict the severity of a buggy change.

General Perform more industrial experiments.

General Combine statistical methods of bug prediction with program analysis.

8.3 Human Feedback

Increasing the permitted types of human feedback Currently, chapter 7 focuses on active learning based feedback mechanism to correct the original label of a code change (buggy or clean). The human feedback is delivered to the Fix Matrix and can modify fix suggestions. However, it would be useful to consider direct feedback on the fix suggestions. A challenge faced when trying this out is the sparsity of bug and fix data. It's not clear what the modifications to a classifier have to occur when certain fix suggestions are incorrect. Without too many similar code changes, it is hard to modify the classifier. Even if there are many similar changes, it is unclear on how exactly the fix matrix should be modified to continue working with reasonable accuracy.

A possible idea to investigate is to display similar code changes and decide if the fix suggestions are invalid for the similar changes as well. An alternative solution is to extract rules for a set of code changes from the feedback.

Providing automated mechanisms to validate human feedback Chapter 7 assumes that human feedback is always accurate. This is also the norm in ML and IR research. It is however possible that with a lot of training data, human feedback might make results worse. Mechanisms to validate or de-prioritize human

feedback should be investigated.

8.4 Fix Suggester

Refining the Fix Matrix to hold complex relationships The fix matrix presented in section 6.4.2.1 forms a linear term frequency relationship between buggy and fix terms. More complex models exploiting relations between buggy and fix terms should be tried. Simple enhancements include techniques to filter out correlated bug terms (and possibly fix terms). More complex enhancements include considering non linear relationships between certain buggy and fix terms, perhaps amongst the popular buggy and fix term pairs. Perhaps text mining techniques such as LDA [30] can be exploited to improve the association between buggy changes and their fixes.

8.5 Bug Prognosticator

Predicting the severity of a buggy change The Bug Prognosticator discussed in chapter 5 does not distinguish bugs by severity. The performance of the predictor is judged by its utility in predicting a buggy change correctly, whether it is a minuscule or major issue.

It is worthy to investigate if bug prediction can also return severities for bugs. A naive approach might construct multiple classifiers, each of which can predict buggy changes of a certain severity. One can then weight the solution provided by

multiple classifiers. This method can work if a clear link to historical link to bug severity can be located. If not, predicting severity can be a much harder problem.

A more refined approach could use a classifier such as the multi-class SVM [162]. A multi-class SVM can be used to not only return if a code change is buggy but also provide a severity. This assumes that bugs of every severity are present in mass numbers in a bug repository.

8.6 General

Combine statistical methods of bug prediction with program analysis If statistical methods of bug prediction (and fix suggestion) can be combined with program analysis techniques, one can potentially have the best of both worlds. For example, if the Fix Suggester has a set of suggested tokens, it would be useful to see if tools such as Findbugs and Coverity agree on the keywords. If so, one can provide suggestions from these tools with the insight that both statistical and static analysis tools are addressing the same solution. As already mentioned, statistical methods reflect more practical bugs whereas program analysis covers cases of well known bug patterns. Merging the two approaches can lead to known solutions to practical bugs that need to be fixed. A qualitative study on the Fix Suggester (section 7.4.2) has indicated human interest in fix suggestions which are filtered using static analysis.

Perform more industrial experiments One commercial project was analyzed in

chapter 5. While chapter 7 leveraged feedback from industrial practitioners, it was restricted to open source projects in order to sidestep challenges from reviewing proprietary code. Industrial validation is the ultimate goal of any software research. More projects utilizing bug prediction and fix suggestion in industrial settings will benefit both research and practice.

The next chapter concludes the dissertation.

Chapter 9

Conclusions

This dissertation tackles the problems of bugs during a code change. Contributions were made on:

Bug Prognosticator A method to improve bug prediction at the code change level.

Fix Suggester A technique that statistically predicts fix tokens for a buggy code change.

Human Feedback A mechanism to improve the results of the Bug Prognosticator and Fix Suggester using human feedback. By virtue of feedback, users of the system will also better understand statistical insights of the above contributions.

A qualitative evaluation of the Fix Suggester.

Chapter 5 introduces the Bug Prognosticator. This solution improves code change prediction to high levels of precision and recall of 0.97 and 0.7 respectively. There is 97% likelihood of a code change being buggy if predicted. About 30% of buggy

code changes are missed. Despite the latter figure, these numbers mean the technique is quite usable in practice.

The feature selection process of chapter 5 is also generally applicable. For example, it was used in [91] to predict crash-prone methods in code.

Chapter 6 introduces the Fix Suggester. While the Fix Suggester sports a precision and recall of 46.9% and 38.9% respectively. While these numbers seem high for the tough area of fix content prediction, the findings need further work to confirm if the presented tokens can eventually help developers develop a bug fix faster.

Improvements to the above contributions using human feedback were detailed in chapter 7. A substantial increase in the Fix Suggester's performance was observed when feedback was conducted on just ten revisions. Feedback can also increased human involvement with the Bug Prognosticator and the Fix Suggester. A qualitative study on the Fix Suggester's output was also performed in chapter 7. While there is scope for improvement, users typically found the Fix Suggester to be useful in practice.

Statistical bug prediction techniques have reached practical significance. It should only be a matter of time before software developers get statistical bug prediction techniques integrated into their software development environment. The use of the Bug Prognosticator will help catch defects as soon as they are created. The Fix Suggester will help developers construct bug fixes to existing bugs. Finally, human feedback will empower developers to be actively involved in statistical defect prediction as opposed to being a passive observer. The golden age of bug and fix predicting imps will have arrived.

Bibliography

[1] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta. Agile Software Development Methods. *Vtt Publications*, 2002.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. An Evaluation of Similarity Coefficients for Software Fault Localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, pages 39–46, Washington, DC, USA, 2006.

[3] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. An Observation-based Model for Fault Localization. In *Proceedings of the International Workshop on Dynamic Analysis*, pages 64–70, 2008.

[4] A. Ahmad and L. Dey. A Feature Selection Technique for Classificatory Analysis. *Pattern Recognition Letters*, 26(1):43–56, 2005.

[5] E. Alpaydin. *Introduction To Machine Learning.* MIT Press, 2004.

[6] Glenn Ammons, Rastislav Bodk, and James R Larus. Mining Specifications. *SIGPLAN Notices*, 37(1):4–16, 2002.

[7] A. Anagnostopoulos, A. Broder, and K. Punera. Effective and Efficient Classification on a Search-Engine Model. *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*, 2006.

[8] J. H Andrews, T. Menzies, and F. C.H Li. Genetic Algorithms for Randomized Unit Testing. *IEEE Transactions on Software Engineering*, 37(1):80–94, 2011.

[9] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.G. Guéhéneuc. Is it a Bug or an Enhancement? A Text-Based Approach to Classify Change Requests. In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, page 23, 2008.

[10] J. Anvik, L. Hiew, and G.C. Murphy. Who Should Fix this Bug? In *Proceedings of the 28th International Conference on Software Engineering*, pages 361–370, 2006.

[11] Apache Logging Services. http://logging.apache.org/, 2011.

[12] A Arcuri. On the Automation of Fixing Software Bugs. In *Proceedings of the 30th International Conference on Software Engineering*, pages 1003–1006, Leipzig, Germany, 2008.

[13] A Arcuri and L Briand. A Practical Guide for using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1–10, 2011.

[14] A. Arcuri and Xin Y. A Novel Co-evolutionary Approach to Automatic Software

Bug Fixing. In *IEEE Congress on Evolutionary Computation*, pages 162–168, 2008.

[15] White D. Clark J. Arcuri, A. and X. Yao. Multi-objective Improvement of Software using Co-evolution and Smart Seeding. In *Proceedings of the 7th International Conference on Simulated Evolution and Learning*, pages 61–70, Melbourne, Australia, 2008.

[16] M. Askari and R. Holt. Information Theoretic Evaluation of Change Prediction Models for Large-Scale Software. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 126–132, 2006.

[17] AspectJ. http://www.eclipse.org/aspectj/, 2011.

[18] L. Aversano, L. Cerulo, and C. Del Grosso. Learning from Bug-introducing Changes to Prevent Fault Prone Code. In *Proceedings of the Foundations of Software Engineering*, pages 19–26, 2007.

[19] N. Ayewah, D. Hovemeyer, J.D. Morgenthaler, J. Penix, and W. Pugh. Using Static Analysis to Find Bugs. *IEEE Journal of Software*, 25(5):22–29, 2008.

[20] Djuradj Babich, Peter J. Clarke, James F. Power, and B. M. Golam Kibria. Using a Class Abstraction Technique to Predict Faults in OO Classes: a Case Study Through Six Releases of the Eclipse JDT. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1419–1424, 2011.

[21] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and

Abraham Bernstein. The Missing Links: Bugs and Bug-Fix Commits. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 97–106, 2010.

[22] BBC News. Microsoft Zune Affected by 'Bug'. `http://news.bbc.co.uk/2/hi/technology/7806683.stm`, 2008.

[23] A. Bernstein, J. Ekanayake, and M. Pinzger. Improving Defect Prediction using Temporal Features and Non Linear Models. In *Proceedings of the 9th International Workshop on Principles of Software Evolution: in conjunction with the 6th ESEC/FSE joint meeting*, pages 11–18, 2007.

[24] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Communications of the ACM*, 53(2):66–75, 2010.

[25] J. Bevan, E.J. Whitehead Jr, S. Kim, and M. Godfrey. Facilitating Software Evolution Research with Kenyon. *Proceedings of joint 10th conference on ESEC/FSE 2005*, pages 177–186, 2005.

[26] Jennifer Bevan, E. James Whitehead, Jr., Sunghun Kim, and Michael Godfrey. Facilitating Software Evolution Research with Kenyon. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering conference held jointly with*

13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 177–186, 2005.

[27] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and Balanced?: Bias in Bug-Fix Datasets. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE '09, pages 121–130, 2009.

[28] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and Balanced?: Bias in Bug-fix Datasets. In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 121–130, 2009.

[29] ZW Birnbaum and F.H. Tingey. One-sided Confidence Contours for Probability Distribution Functions. *The Annals of Mathematical Statistics*, pages 592–596, 1951.

[30] D.M. Blei, A.Y. Ng, and M.I. Jordan. Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[31] B. Boehm and V.R. Basili. Defect Reduction Top 10 List. *IEE Computer*, pages 135–137, 2001.

[32] B.W. Boehm and P.N. Papaccio. Understanding and Controlling Software Costs. *IEEE Transactions on Software Engineering*, 14(10):1462–1477, 1988.

[33] J.F. Bowring, J.M. Rehg, and M.J. Harrold. Active Learning for Automatic Classification of Software Behavior. *ACM SIGSOFT Software Engineering Notes*, 29(4):195–205, 2004.

[34] Lionel C. Briand, Jurgen Wiist, Stefan V. Ikonomovski, and Hakim Lounis. Investigating Quality Factors in Object-oriented Designs: An Industrial Case Study. *International Conference on Software Engineering*, 0:345–354, 1999.

[35] Y. Brun and M. Ernst. Finding Latent Code Errors via Machine Learning over Program Executions. *Proceedings of the 26th International Conference on Software Engineering*, pages 480–490, 2004.

[36] JD Cem Kaner, E. Hendrickson, and J. Smith-Brock. Managing the Proportion of Testers to (Other) Developers. *Quality Week*, 2001.

[37] V.U.B. Challagulla, F.B. Bastani, I.L. Yen, and R.A. Paul. Empirical Assessment of Machine Learning Based Software Defect Prediction Techniques. In *Proceedings of the 10th International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 263–270. IEEE, 2005.

[38] O. Chapelle, B. Schölkopf, A. Zien, and I. NetLibrary. *Semi-supervised Learning*. MIT Press, 2006.

[39] Holger Cleve and Andreas Zeller. Locating Causes of Program Failures. In *Proceedings of the 27th International Conference on Software Engineering*, pages 342–351, 2005.

[40] W.W. Cohen. Fast Effective Rule Induction. *Proceedings of the 12th International Conference on Machine Learning*, pages 115–123, 1995.

[41] DA Cohn, Z. Ghahramani, and MI Jordan. Active Learning with Statistical Models. *Arxiv preprint cs.AI/9603104*, 1996.

[42] Barthélémy Dagenais and Martin P. Robillard. Recommending Adaptive Changes for Framework Evolution. In *Proceedings of the 30th International Conference on Software Engineering*, pages 481–490, 2008.

[43] V. Dallmeier, A. Zeller, and B. Meyer. Generating Fixes from Object Behavior Anomalies. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 550–554, 2009.

[44] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining Object Behavior with ADABU. In *Proceedings of the International Workshop on Dynamic Systems Analysis*, pages 17–24, Shanghai, China, 2006.

[45] Valentin Dallmeier and Thomas Zimmermann. Extraction of Bug Localization Benchmarks from History. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 433–436, 2007.

[46] M. D'Ambros, M. Lanza, and R. Robbes. Evaluating Defect Prediction Approaches: a Benchmark and an Extensive Comparison. *Empirical Software Engineering*, pages 1–47, 2011.

[47] Brian Demsky, Michael D Ernst, Philip J Guo, Stephen McCamant, Jeff H Perkins, and Martin Rinard. Inference and Enforcement of Data Structure Consistency Specifications. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 233–244, Portland, Maine, USA, 2006.

[48] Brian Demsky and Martin Rinard. Data Structure Repair Using Goal-directed Reasoning. In *Proceedings of the 27th International Conference on Software Engineering*, pages 176–185, St. Louis, MO, USA, 2005.

[49] A. Eastwood. Firm Fires Shots at Legacy Systems. *Computing Canada*, 19(2):17, 1993.

[50] Eclipse Java Development Tools. http://www.eclipse.org/jdt/, 2011.

[51] B. Efron and R. Tibshirani. *An Introduction to the Bootstrap*, volume 57. Chapman & Hall/CRC, 1993.

[52] Agoston E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing.* Springer, 2008.

[53] K. Elish and M. Elish. Predicting Defect-Prone Software Modules Using Support Vector Machines. *Journal of Systems and Software*, 81(5):649–660, 2008.

[54] Len Erlikh. Leveraging Legacy System Dollars for E-business. *IT Professional*, 2(3):17–23, 2000.

[55] R.E. Fan, K.W. Chang, C.J. Hsieh, X.R. Wang, and C.J. Lin. Liblinear: A Library for Large Linear Classification. *The Journal of Machine Learning Research*, 9:1871–1874, 2008.

[56] T. Fawcett. An Introduction to ROC Analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006.

[57] B. Fluri and H.C. Gall. Classifying Change Types for Qualifying Change Couplings. *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 35–45.

[58] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[59] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A Genetic Programming Approach to Automated Software Repair. In *Proceedings of the 11th annual Conference on Genetic and Evolutionary Computation*, pages 947–954, 2009.

[60] J. Friedman, T. Hastie, and R. Tibshirani. *The Elements of Statistical Learning*, volume 1. Springer Series in Statistics, 2001.

[61] Mark Gabel and Zhendong Su. Symbolic Mining of Temporal Specifications. In

Proceedings of the 30th International Conference on Software Engineering, pages 51–60, Leipzig, Germany, 2008.

[62] K. Gao, T.M. Khoshgoftaar, H. Wang, and N. Seliya. Choosing Software Metrics for Defect Prediction: an Investigation on Feature Selection Techniques. *Software: Practice and Experience*, 41(5):579–606, 2011.

[63] M. Gegick, P. Rotella, and T. Xie. Identifying Security Bug Reports via Text Mining: An Industrial Case Study. In *Proceedings of the 7th Working Conference on Mining Software Repositories*, pages 11–20. IEEE, 2010.

[64] Carlo Ghezzi, Andrea Mocci, and Mattia Monga. Efficient Recovery of Algebraic Specifications for Stateful Components. In *9th International Workshop in conjunction with the 6th ESEC/FSE joint meeting on Principles of Software Evolution*, pages 98–105, Dubrovnik, Croatia, 2007.

[65] T. Glasmachers and C. Igel. Second-order Smo Improves Svm Online and Active Learning. *Neural Computation*, 20(2):374–382, 2008.

[66] Claire Goues and Westley Weimer. Specification Mining with Few False Positives. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, pages 292–306, York, UK, 2009.

[67] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy. Predicting Fault Incidence

Using Software Change History. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[68] James Grenning. Zune Bug: Test Driven Bug Fix. `http://www.renaissancesoftware.net/blog/archives/38`, 2009.

[69] T. Gyimóthy, R. Ferenc, and I. Siket. Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2005.

[70] M.A. Hall and G. Holmes. Benchmarking Attribute Selection Techniques for Discrete Class Data Mining. *IEEE Transactions on Knowledge and Data Engineering*, 15(6):1437–1447, 2003.

[71] A Hassan. Mining Software Repositories to Assist Developers and Support Managers. *Software Maintenance*, 2006.

[72] A. Hassan and R. Holt. The Top Ten List: Dynamic Fault Prediction. *Proceedings of the 21st IEEE International Conference on Software Maintenance*, 2005.

[73] H. Hata, O. Mizuno, and T. Kikuno. An Extension of Fault-prone Filtering using Precise Training and a Dynamic Threshold. *Proceedings of the international working conference on Mining Software Repositories*, 2008.

[74] Brian Hayes. The Zune Bug. `http://bit-player.org/2009/the-zune-bug`, 2009.

[75] R. Holmes and G.C. Murphy. Using Structural Context to Recommend Source Code Examples. In *Proceedings of the 27th International Conference on Software Engineering*, pages 117–125, 2005.

[76] Scientific Toolworks. http://www.scitools.com/. Maintenance, Understanding, Metrics and Documentation Tools for Ada, C, C++, Java, and Fortran. 2005.

[77] S. Huff. Information Systems Maintenance. *The Business Quarterly*, 55:30–32, 1990.

[78] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K Chang. OCAT: Object Capture-based Automated Testing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pages 159–170, Trento, Italy, 2010.

[79] T. Joachims. Text Categorization with Support Vector Machines: Learning with Many Relevant Features. *Machine Learning: ECML*, pages 137–142, 1998.

[80] T. Joachims. Training Linear SVMs in Linear Time. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 226, 2006.

[81] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering*, pages 467–477, 2002.

[82] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of Test

Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering*, pages 467–477, Orlando, Florida, 2002.

[83] H. Joshi, C. Zhang, S. Ramaswamy, and C. Bayrak. Local and Global Recency Weighting Approach to Bug Prediction. *Proceedings of the 4th International Workshop on Mining Software Repositories*, 2007.

[84] G. Kasparov. The Chess Master and the Computer. *The New York Review of Books*, 57(2):16–19, 2010.

[85] T. Khoshgoftaar and E. Allen. Predicting the Order of Fault-Prone Modules in Legacy Software. *Proceedings of the International Symposium on Software Reliability Engineering*, pages 344–353, 1998.

[86] T. Khoshgoftaar and E. Allen. Ordering Fault-Prone Software Modules. *Software Quality J.*, 11(1):19–37, 2003.

[87] S. Kim, E.J. Whitehead Jr., and Y. Zhang. Classifying Software Changes: Clean or Buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.

[88] S. Kim, T. Zimmermann, E.J. Whitehead Jr., and Andreas Zeller. Predicting Faults from Cached History. *Proceedings of the 29th International Conference on Software Engineering*, pages 489–498, 2007.

[89] S. Kim, T. Zimmermann, E.J. Whitehead Jr, and A. Zeller. Predicting Faults from Cached History. In *Proceedings of the 29th International conference on Software Engineering*, pages 489–498. IEEE Computer Society, 2007.

[90] Sunghun Kim, Kai Pan, and E. E. James Whitehead, Jr. Memories of Bug Fixes. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 35–45, 2006.

[91] Sunghun Kim, Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, and Shivkumar Shivaji. Predicting Method Crashes with Bytecode Operations. In *Indian Software Engineering Conference*, pages 3–12, 2013.

[92] P. Knab, M. Pinzger, and A. Bernstein. Predicting Defect Densities in Source Code Files with Decision Tree Learners. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 119–125, 2006.

[93] I. Kononenko. Estimating Attributes: Analysis and Extensions of Relief. In *European Conference on Machine Learning and Principles*, pages 171–182. Springer, 1994.

[94] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, 1st edition, 1992.

[95] R. Kumar, S. Rai, and J. Trahan. Neural-Network Techniques for Software-Quality Evaluation. *Reliability and Maintainability Symposium*, 1998.

[96] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals. Predicting the Severity of a Reported Bug. In *Proceedings of the 7th Working Conference on Mining Software Repositories*, pages 1–10. IEEE, 2010.

[97] J. Langford, L. Li, and T. Zhang. Sparse Online Learning via Truncated Gradient. *The Journal of Machine Learning Research*, 10:777–801, 2009.

[98] B. Larsen and C. Aone. Fast and Effective Text Mining using Linear-time Document Clustering. In *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 16–22, 1999.

[99] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. *IEEE Transactions on Software Engineering*, 34(4):485–496, 2008.

[100] D. Lewis. Naive (Bayes) at Forty: The Independence Assumption in Information Retrieval. *Machine Learning: ECML*, pages 4–15, 1998.

[101] Ben Liblit. Building a Better Backtrace: Techniques for Postmortem Program Analysis. Technical report, Computer Science Division, University of California, 2002.

[102] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable Statistical Bug Isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 15–26, 2005.

[103] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. Sober: Statistical Model-based Bug Localization. In *Proceedings of the 10th European*

Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 286–295, 2005.

[104] H. Liu and H. Motoda. *Feature Selection for Knowledge Discovery and Data Mining*. Springer, 1998.

[105] Rick Lockridge. Will Bugs Scare off Users of New Windows 2000. `http://articles.cnn.com/2000-02-17/tech/windows.2000_1_microsoft-memo-windows-nt-software-washingtonbased-microsoft?_s=PM:TECH`, 2000.

[106] Lucia Lucia, David Lo, Lingxiao Jiang, and Aditya Budi. Active Refinement of Clone Anomaly Reports. In *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[107] J.T. Madhavan and E.J. Whitehead Jr. Predicting Buggy Changes Inside an Integrated Development Environment. *Proceedings of the Eclipse Technology eXchange*, 2007.

[108] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. PSE: Explaining Program Failures via Postmortem Static Analysis. *SIGSOFT Software Engineering Notes*, 29(6):63–72, 2004.

[109] H. B. Mann. On a Test of Whether One of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947.

[110] F.J. Massey Jr. The Kolmogorov-Smirnov Test for Goodness of Fit. *Journal of the American Statistical Association*, pages 68–78, 1951.

[111] A. McCallum and K. Nigam. A Comparison of Event Models for Naive Bayes Text Classification. *Workshop on Learning for Text Categorization*, 1998.

[112] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald. Problems with Precision: A Response to "Comments on Data Mining Static Code Attributes to Learn Defect Predictors". *IEEE Transactions on Software Engineering*, 33(9):637–640, 2007.

[113] T. Menzies, J. Greenwald, and A. Frank. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Trans. Software Eng.*, 33(1):2–13, 2007.

[114] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Yi Wei, and E. Stapf. Programs That Test Themselves. *Computer*, 42(9):46–55, 2009.

[115] O Mizuno and T Kikuno. Training on Errors Experiment to Detect Fault-Prone Software Modules by Spam Filter. *Proceedings of the 11th Joint Meeting of the European Software Engineering Conference and the 14th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 405–414, 2007.

[116] A. Mockus and L. Votta. Identifying Reasons for Software Changes using Historic Databases. *Proceedings of the 16th IEEE International Conference on Software Maintenance*, page 120, 2000.

[117] A. Mockus and D. Weiss. Predicting Risk of Software Changes. *Bell Labs Techical Journal*, 5(2):169–180, 2000.

[118] Sandro Morasca and Günther Ruhe. A Hybrid Approach to Analyze Empirical Software Engineering Data and its Application to Predict Module Fault-proneness in Maintenance. *Journal of Systems Software*, 53(3):225–237, 2000.

[119] H. Moravec. When will Computer Hardware Match the Human Brain. *Journal of Evolution and Technology*, 1(1):10, 1998.

[120] R. Moser, W. Pedrycz, and G. Succi. A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction. In *Proceedings of the 30th International Conference on Software Engineering*, pages 181–190. IEEE, 2008.

[121] J. Mühlberg and G. Lüttgen. Blasting Linux Code. *Formal Methods: Applications and Technology*, pages 211–226, 2007.

[122] Erica Naone. So Many Bugs, So Little Time. `http://www.technologyreview.com/news/419975/so-many-bugs-so-little-time/`, 2010.

[123] National Institute of Standards and Technology. Software Errors Cost U.S. Economy $59.5 Billion Annually. `http://www.abeacha.com/NIST_press_release_bugs_cost.htm`, 2002.

[124] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting Vulnerable

Software Components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 529–540, 2007.

[125] T.T. Nguyen, H.A. Nguyen, N.H. Pham, J. Al-Kofahi, and T.N. Nguyen. Recurring Bug Fixes in Object-oriented Programs. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 315–324, 2010.

[126] T.J. Ostrand, E.J. Weyuker, and R.M. Bell. Predicting the Location and Number of Faults in Large Software Systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

[127] Ovum. Ovum Software Market Forecast, 2011.

[128] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, 2007.

[129] Alan Page. http://blogs.msdn.com/b/alanpa/archive/2007/08/24/shipping-with-bugs.aspx, 2007.

[130] K. Pan, S. Kim, and E. Whitehead Jr. Bug Classification Using Program Slicing metrics. *Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation*, pages 31–42, 2006.

[131] Kai Pan, Sunghun Kim, and E. James Whitehead. Toward an Understanding of Bug Fix Patterns. *Empirical Software Engineering*, 14(3):286–315, 2008.

[132] J.R. Quinlan. *C4. 5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[133] Rhino: JavaScript for Java. http://www.mozilla.org/rhino/, 2011.

[134] M. Rinard, C. Cadar, D. Dumitran, D. M Roy, and T. Leu. A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities (and other Memory Errors). In *Computer Security Applications Conference, 2004. 20th Annual*, pages 82–90, 2004.

[135] M. Robnik-Šikonja and I. Kononenko. Theoretical and Empirical Analysis of Relieff and RRelieff. *Machine Learning*, 53(1):23–69, 2003.

[136] T. Rölleke, T. Tsikrika, and G. Kazai. A General Matrix Framework for Modelling Information Retrieval. *Information Processing and Management*, 42(1):4–30, 2006.

[137] W.W. Royce. Managing the Development of Large Software Systems. In *Proceedings of IEEE WESCON*, volume 26. Los Angeles, 1970.

[138] N. Rutar, C.B. Almazan, and J.S. Foster. A Comparison of Bug Finding Tools for Java. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 245–256. IEEE, 2004.

[139] S.R. Safavian and D. Landgrebe. A Survey of Decision Tree Classifier Methodology. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(3):660–673, 1991.

[140] C. Saunders, M.O. Stitson, J. Weston, L. Bottou, A. Smola, et al. Support Vector Machine Reference Manual. *Royal Holloway University, London, Techical Report CSD-TR-98-03*, 1998.

[141] A. Schröter, T. Zimmermann, and A. Zeller. Predicting Component Failures at Design Time. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 18–27, 2006.

[142] S. Scott and S. Matwin. Feature Engineering for Text Classification. *Machine Learning-International Workshop*, pages 379–388, 1999.

[143] E. Shihab, Z.M. Jiang, W.M. Ibrahim, B. Adams, and A.E. Hassan. Understanding the Impact of Code and Process Metrics on Post-Release Defects: a Case Study on the Eclipse Project. *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2010.

[144] S. Shivaji, E.J. Whitehead Jr, R. Akella, and S. Kim. Reducing Features to Improve Bug Prediction. In *International Conference on Automated Software Engineering*, pages 600–604. IEEE/ACM, 2009.

[145] S. Shivaji, E.J. Whitehead Jr, R. Akella, and S. Kim. Reducing Features to Improve Code Change Based Bug Prediction. *IEEE Transactions on Software Engineering*, 2012.

[146] F. Shull, V. Basili, B. Boehm, A.W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz. What We Have Learned about Fighting Defects. In *Proceedings of the 8th IEEE Symposium on Software Metrics*, pages 249–258. IEEE, 2002.

[147] Sink, Eric. Why We All Sell Code with Bugs. `http://www.guardian.co.`

uk/technology/2006/may/25/insideit.guardianweeklytechnologysection, 2006.

[148] John Sinteur. Cause of Zune 30 Leapyear Problem Isolated! http://www.zuneboards.com/forums/zune-news/38143-cause-zune-30-leapyear-problem-isolated.html, 2008.

[149] J. Sliwerski, T. Zimmermann, and A. Zeller. When Do Changes Induce Fixes? *Proc. MSR 2005*, pages 24–28, 2005.

[150] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu. A General Software Defect-proneness Prediction Framework. *IEEE Transactions on Software Engineering*, (99):356–370, 2011.

[151] M. Tang, X. Luo, and S. Roukos. Active learning for statistical natural language parsing. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 120–127. Association for Computational Linguistics, 2002.

[152] R. Telang and S. Wattal. An Empirical Analysis of the Impact of Software Vulnerability Announcements on Firm Stock Price. *IEEE Transactions on Software Engineering*, 33(8):544–557, 2007.

[153] S. Tong and D. Koller. Support vector machine active learning with applications to text classification. *The Journal of Machine Learning Research*, 2:45–66, 2002.

[154] C. J. van Rijsbergen. *Information Retrieval*. Butterworth, 1979.

[155] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb. An Evaluation of Two Bug Pattern Tools for Java. In *1st International Conference on Software Testing, Verification, and Validation*, pages 248–257. IEEE, 2008.

[156] F. Wedyan, D. Alrmuny, and J.M. Bieman. The Effectiveness of Automated Static Analysis Tools for Fault Detection and Refactoring Prediction. In *International Conference on Software Testing Verification and Validation*, pages 141–150. IEEE, 2009.

[157] Yi Wei, Carlo A Furia, Nikolay Kazmin, and Bertrand Meyer. Inferring Better Contracts. In *Proceeding of the 33rd International Conference on Software Engineering*, pages 191–200, Waikiki, Honolulu, HI, USA, 2011.

[158] Yi Wei, Yu Pei, Carlo A Furia, Lucas S Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated Fixing of Programs with Contracts. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pages 61–72, Trento, Italy, 2010.

[159] Westley Weimer. Patches as Better Bug Reports. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, pages 181–190, Portland, Oregon, USA, 2006.

[160] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically Finding Patches using Genetic Programming. In *Proceedings of the*

31st International Conference on Software Engineering, pages 364–374, Washington, DC, USA, 2009.

[161] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How Long will it take to Fix this Bug? In *Proceedings of the 4th International Workshop on Mining Software Repositories*, page 1. IEEE Computer Society, 2007.

[162] Jason Weston and Chris Watkins. Support Vector Machines for Multi-class Pattern Recognition. In *Proceedings of the 7th European Symposium on Artificial Neural Networks*, volume 4, pages 219–224, 1999.

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

[164] C Williams and J Hollingsworth. Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques. *IEEE Transactions on Software Engineering*, 31(6):466–480, 2005.

[165] Hyrum K. Wright, Miryung Kim, and Dewayne E. Perry. Validity Concerns in Software Engineering Research. In *Proceedings of the FSE/SDP workshop on Future of Software Engineering Research*, FoSER '10, pages 411–414, 2010.

[166] G. Xiao, F. Southey, R.C. Holte, and D. Wilkinson. Software Testing by Active Learning for Commercial Games. In *Proceedings of the National Conference on Artificial Intelligence*, volume 2, pages 898–903. AAAI Press, 2005.

[167] S. Zaman, B. Adams, and A.E. Hassan. Security versus Performance Bugs: A

Case Study on Firefox. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 93–102, 2011.

[168] Andreas Zeller. Yesterday, My Program Worked. Today, It Does Not. why? In *Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, volume 24, pages 253–267, 1999.

[169] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2005.

[170] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating Faults through Automated Predicate Switching. In *Proceedings of the 28th International Conference on Software Engineering*, pages 272–281, 2006.

[171] T. Zimmermann, R. Premraj, and A. Zeller. Predicting Defects for Eclipse. In *Proceedings of the International Workshop on Predictor Models in Software Engineering*. IEEE, 2007.

[172] T. Zimmermann and P. Weißgerber. Preprocessing CVS Data for Fine-Grained Analysis. *Proceedings of the International Workshop on Mining Software Repositories*, pages 2–6, 2004.