UNIVERSITY OF CALIFORNIA

SANTA CRUZ

ADAPTIVE BUG PREDICTION BY ANALYZING PROJECT HISTORY

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

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Sunghun Kim

September 2006

The Dissertation of Sunghun Kim
is approved:

E. James Whitehead, Jr.
Professor E. James Whitehead, Jr. (Chair)

David P. Helmbold
Professor David P. Helmbold

[illegible]
Professor Cormac Flanagan

Lisa C. Sloan
Vice Provost and Dean of Graduate Studies

Contents

List of Figures

List of Tables

Abstract

ADAPTIVE BUG PREDICTION BY ANALYZING PROJECT HISTORY

by

Sunghun Kim

Finding and fixing software bugs is difficult, and many developers put significant effort into finding and fixing them. A project's software change history records the change that introduces a bug and the change that subsequently fixes it. This bug-introducing and bug-fix experience can be used to predict future bugs. This dissertation presents two bug prediction algorithms that adaptively analyze a project's change history: bug cache and change classification.

The basic assumption of the bug cache approach is that the bugs do not occur in isolation, but rather in a burst of several related bugs. The bug cache exploits this locality by caching locations that are likely to have bugs. By consulting the bug cache, a developer can detect locations likely to be fault prone. This is useful for prioritizing verification and validation resources on the most bug prone files, functions, or methods. An evaluation of seven open source projects with more than

200,000 revisions shows that the bug cache selects 10% of the source code files; these files account for 73%-95% of future bugs.

The change classification approach learns from previous buggy change patterns using two machine learning algorithms, Naïve Bayes and Support Vector Machine. After training on buggy change patterns, it predicts new unknown changes as either buggy or clean. As soon as changes are made, developers can use the predicted information to inspect the new changes, which are an average of 20 lines of code. After training on 12 open source projects, the change classification approach can, on average, classify buggy changes with 78% accuracy and 65% buggy change recall.

By leveraging project history and learning the unique bug patterns of each project, both approaches can be used to find locations of bugs. This information can be used to increase software quality and reduce software development cost.

Acknowledgements

I am very lucky to have met my advisor, Jim Whitehead. He offered me advice on potential research topics and helped guide me through my research. He was kind and always encouraging. His great talent in understanding my ideas helped direct me along the right path. He provided me with advice for, not only my research, but personal issues as well. Whenever I had issues in my life, I didn’t hesitate to bring them to him. Without his guidance, it would not have been possible to perform my research and finish my Ph.D.

My wife, Yeon, has been with me throughout my entire graduate career. She provides me with a tremendous amount of love and support. Whenever I was downhearted, she encouraged and provided me with the power to overcome my hardships. With her love and support I can enjoy my Ph.D life at Santa Cruz. My family in Korea, Mom, Dad, Jong-Suk, Jong-Chul, Yeon-Sun, Yeon-Hee have also provided support and encouragement. Yong-Doo Lee, the president of Daegu University encouraged me. I would like to thank Haeam members for their friendship.

Kai Pan is the best lab-mate one could ever have. He is very smart and knowledgeable on our research, while still being kind and humble. Whenever I faced some obstacle in my research, we discussed and solved the problems together. Guozheng Ge provided me with good research and friendship. His comments and opinions about my research were sharp and provided me with good insight. I would like to thank Mark Slater, Jennifer Bevan, and Elias Sinderson for their support and friendship as well.

Yi Zhang, David Helmbold, and Cormac Flanagan enriched my dissertation by providing me with good comments, insight, and questions.

I was lucky to have very good collaborators, Thomas Zimmermann, Andreas Zeller, and Miryung Kim. Even though they were in Germany and Washington, they encouraged me and provided me with good comments on my research.

Santa Cruz Korean community members, Chris, Dong-Pyo, Shin-Ji, Kyung-Jin, Mi-Suk, Hee-Jin, Mu-Sung, Jung-Eun were like family to me. They cared for and prayed for me. With their love and support I have had a blessed graduate career in Santa Cruz.

I would like to thank the Storage Systems Research Center and Kevin Greenan for letting me use their cluster machines. My research included many experiments with various software projects. By using their cluster machines, I was able to finish all necessary experiments in time.

Finally I would like to thank Sheena Marie Marquez, who revised my dissertation and gave me good comments.

To my wife

1 Introduction

In 1945, a moth became trapped between the points of a relay in the Mark II Aiken Relay Calculator, causing the computer to perform incorrectly; this was the first computer bug [25]. Today, despite tremendous advances in computer hardware and software technologies there are still many latent bugs in software.

Software bugs can cause a range of problems, ranging from minor glitches to loss of life or significant material loss [18]. For example, in 1962 a flight control software bug caused a rocket to divert from its intended path on launch, which led mission control to destroy the rocket over the Atlantic Ocean [18]. In 1996, due to a software bug, the European Space Agency's US $1 billion prototype Ariane 5 rocket was destroyed right after launch [1]. In 1985, the Therac-25 radiation therapy device malfunctioned and delivered lethal radiation doses. This was due to a software bug. In 1988, more than 2,000 computers were infected by the first computer worm, which was made possible because of a buffer overflow bug in the Berkeley Unix finger daemon program [18].

The process of finding and removing bugs is called debugging, a frequent, tedious, and time-consuming task for software developers. In some cases, developers spend more time debugging than writing new source code [1, 66]. Many bugs are not discovered even after an intensive debugging process, with unpredictable and sometimes serious consequences. Current software development practice involves a process of triage on existing uncovered bugs, fixing those of highest priority, with large test suites designed to discover latent bugs. A typical software application ships with multiple known bugs, and hence debugging is a perpetual process.

The goal of this research is to help developers in the debugging process, thereby raising software quality and hopefully lowering the cost of software development and maintenance. To meet this goal, this work proposes algorithms to locate or predict latent bugs in functions, files, and changes by mining project history data.

The traditional debugging process includes code review, using debugging tools such as GDB [15], regression testing, unit testing, and code review again. Locating and predicting a bug is helpful to developers since it can narrow down the debugging scope. For example, instead of running all test cases and verifying correctness for the entire software project, using the techniques in this dissertation, developers receive lists of likely buggy files, methods, or changes. This reduces the scope of software that needs to be examined for bugs down to a single file, method, or change. This allows the developer to run just a small set of test cases that are related to the buggy locations.

The key insight behind this dissertation is learning from software evolution history. Most software uses software configuration management (SCM) systems to manage and record the evolution of a software project. Recorded evolution data includes change history, change log messages, bug occurrences, and bug-fixes. Since each project's evolution data includes years of bug-fix experiences, they can be a good resource for predicting bugs, by learning from previous mistakes. Currently, evolution data is not fully leveraged for predicting bugs. The two approaches described herein use software evolution data for future bug prediction.

There are two types of bugs: horizontal and vertical. Horizontal bugs are general, and occur across all software projects. Vertical bugs are project-specific, and are only found in a single project. For example, consider the null-dereferencing bug example in Figure 1-1. This code pattern is a bug, no matter where it is found, and hence it exemplifies the class of horizontal bugs.

```
if (bar==null) {
  System.out.println(bar.foo);
}
```

Figure 1-1. Example null dereferencing bug.

In contrast, consider two bug-fix examples from the JEdit project, shown in Figure 1-2. In two separate revisions of the same file, use of the '*setSelectText()*' method is considered to be a bug. The corresponding fix is to use the '*insertTab()*'

method. This bug pattern is very project specific, since it only occurs in the JEdit project. Such vertical bugs can only be obtained by analyzing the project's evolution history.

To locate such project specific bugs, a bug prediction model must be project dependent and should be able to learn project specific bug patterns or properties. Such prediction algorithms are called "adaptive" or "dynamic." Even though adaptive prediction models may differ from project to project, the algorithms remain reusable across all projects, and can yield high predictive accuracy.

```
JEditTextArea.java at transaction 86
- setSelectedText("\t");
+ insertTab();
JEditTextArea.java at transaction 114
- setSelectedText("\t");
+ insertTab();
```

Figure 1-2. Repeated bug-fix examples in JEdit. The '–' and highlighted code indicates buggy code and '+ 'indicates the corresponding fix.

Recent work by Nagappan, Ball, and Zeller also indicates that there is no single prediction model that can act as the best defect predictor across projects [47]. As in this dissertation, they observe that a bug prediction model must be trained to grab the unique bug patterns and properties of a specific project.

This dissertation introduces two new bug prediction algorithms: bug cache and change classification. These are adaptive and dynamic bug prediction algorithms that learn from a given project's software evolution history to maximize their predictive power.

The bug cache algorithm identifies those files or functions in a software project that are most likely to have future bugs. The basic assumption of the bug cache is that the occurrence of bugs is not random, but rather local. For example, if a bug occurs in a file or function, this file or function will have another bug soon. Based on the bug localities, the bug cache approach sets a cache for bug prediction. If a bug occurs in a software entity, the bug cache approach loads the entity into the cache on the assumption the entity will have another bug in the near future. The bug cache algorithm predicts future bugs using the list of entities in the cache. If a future bug occurs in an entity in the list, the prediction was correct. Otherwise, the prediction was wrong.

The bug cache is an online learning style approach, in that it maintains the list of the cache based on the prediction results. If the bug prediction is wrong, it is penalized by reconstructing the cache. If the prediction is correct, it is rewarded by keeping the cache as-is. The bug cache can predict about 70%~95% of future bugs at the file level of granularity using 10% cache size. This indicates that developers can focus their debugging efforts on 10% of software files to find 70~95% of the latent bugs.

The change classification approach predicts bugs in changes. It learns a prediction model by mining features from previous buggy and clean changes.

Using the prediction model, it classifies new unknown changes as either buggy or clean. Two machine learning algorithms, Naïve Bayes and Support Vector Machine, are used to train prediction models. The training data for the classifier is obtained from a project's software evolution history. After training a classification model, it can classify changes with 70-93% accuracy.

To evaluate the two approaches, change, bug, and bug-fix history data is mined from 12 open source projects. By mining change logs, bug-fix changes can be identified. Next, the bug-fix changes and revision history are used to identify when bug(s) were introduced into the system which is called a bug-introducing change. Bug-introducing changes are used in training and evaluation of the bug cache and bug classification approaches.

The remainder of the dissertation starts by introducing a software development model and terms used throughout this dissertation (Chapter 2). The method of data extraction from software history is explained in Chapter 3. The two bug prediction approaches, bug cache (Chapter 4) and change classification (Chapter 5) are presented along with their prediction performance and discussion. The limitations of the approaches are discussed in Chapter 6. This dissertation ends with related work (Chapter 7) and conclusions (Chapter 8).

2 Terminology

This chapter describes a common software development process model in terms of the life cycle of a bug, and also defines terms used in this dissertation.

2.1 Fine-Grain software development processes

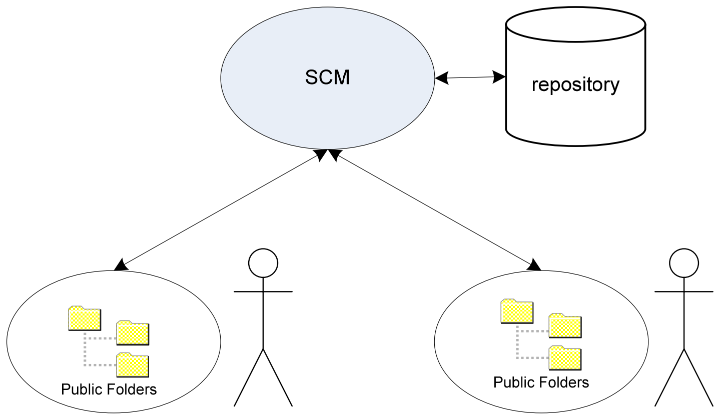

Figure 2-1. Two workspaces and Software Configuration Management (SCM)

File changes are usually stored and managed by software configuration management (SCM) systems. For the open source systems we analyzed, the SCM system used is either CVS [4] or Subversion [3]. Each developer has their own isolated working space called a *workspace* as shown in Figure 2-1. For CVS and Subversion, the workspace is a directory on the developers' local workstation. From the SCM systems, developers *check out* source code into their own workspace. Developers usually change files in their workspace and when they want to store their changes in the SCM system, they submit their changes using the *commit* command. Changes made in workspaces are not visible to other developers until changes are committed to the SCM system, and then each developer updates their local workspace from the SCM system to apply changes from others.

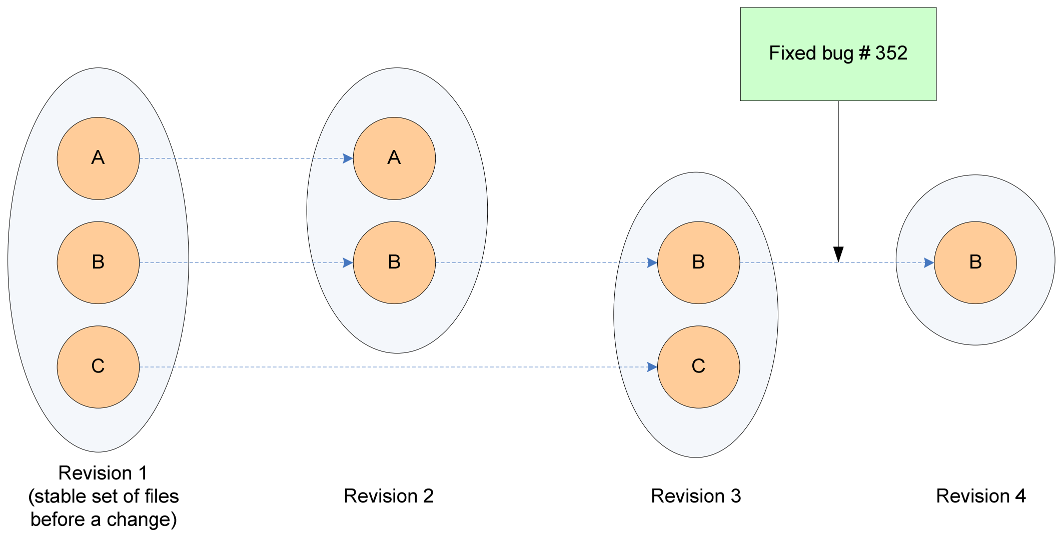

Figure 2-2. Example of a project evolution

When developers commit changes, they can commit more than one file change at the same time. A group of changes at the same time is called a revision or transaction. Suppose there is an evolving software project that consists of three files, A, B, and C, as shown in Figure 2-2. At revision 1, all three files are created and at revision 2, files A and B are changed. Each file modification is called a *change*.

At revision 2 in Figure 2-2, two files have been committed. All of the files changed in the same commit comprise a *transaction*. The changed files in the same transaction are called *co-changed* files. Each transaction has one change log message written by the developer who committed the change. The change log message is used to describe the nature of the changes in the commit, such as “Fixed bug #352”, as shown in Figure 2-2.

During the development process, developers or users can notice some abnormality in the behavior of the software. These abnormal behaviors are often reported by users and developers, and are typically recorded in a bug tracking system such as Bugzilla [61], or on a mailing list. The developers of the software decide if the reported abnormality is a real bug or not. If it is a real bug, developers then try to locate the bug by code inspection, running test cases, static analysis, etc. After locating the bug, developers try to find solutions to correct the bug, called *fixes*. They implement the fix by changing the files that have code related to the bug. When committing a fix, most developers leave a message in the change log indicating that the updated code includes a fix. The revision associated with a fix commit is called a *fix revision*. All file changes in a fix revision are also *fix changes*. For example, revision 4 in Figure 2-2 is a fix revision, and the file change to B in revision 4 is a fix change.

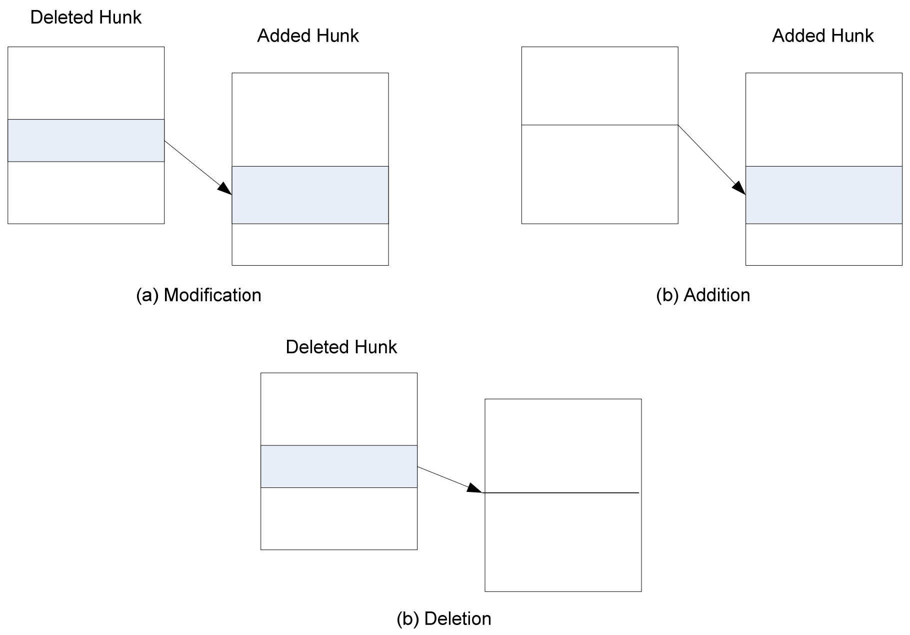

Figure 2-3. Three kinds of changes and hunks.

A fix change consists of two files, which are an old file revision that has a bug and a new file revision that has a bug fixed. There are three types of file changes: modification, addition, and deletion, as shown in Figure 2-3. A group of contiguous changed lines between two files is called a *hunk*. If it is a fix change, the hunks in the old file are called *bug hunks* and in the new file are called *fix hunks*. The bug hunk contains buggy code since, by removing or replacing the hunk, the bug was fixed. By tracing the origins of each line in the bug hunk, the moments of bug creation are revealed [32, 58]. The change that initially created a bug is called a *bug-introducing change*.

The life cycle of a bug is as follows: A bug is created during the development process in a bug-introducing change. Due to the presence of the bug, the software has abnormal behavior that causes users or developers to notice the existence of the bug. Developers locate the bug and then fix it by changing files related to the bug. If we assume the developers fixed the bug correctly, the bug dies.

Traditionally, bugs are identified in software by examining the output from software execution, performing software inspections, or running static analysis tools. The method for bug identification used in this dissertation is somewhat different. Developers are assumed to have been using these traditional methods for bug identification throughout a project's evolution, and have been fixing the buggy code. Based on this assumption, the fixed source code includes bug(s).

2.2 Terms

In this chapter, terms used in this dissertation are defined.

Bug tracking system: Since the appearance of bugs is a serious problem in software, software projects manage lists of bugs and track their status. A bug tracking system stores and manages all bug status and information such as the module where a bug occurred, when a bug was found, the severity of a bug, comments describing a bug's effect on the software, instructions on replicating a bug, who reported the bug, and whether the bug has been fixed yet.

File change (change): Software development proceeds by changing files that are usually stored in an SCM system. A file change is an instance of a file modification stored in an SCM system.

Version: During software evolution, files are changing and have many instances. The term version is used to identify different instances of the same file after a change. For example, '*foo*' file version 1 and '*foo*' file version 2 indicate two different instances of the '*foo*' file after changes.

Commit: Submitting changes to an SCM system is defined as a commit. A commit often includes more than one file change. Developers usually write a change log message when they commit.

Change log: When developers perform a commit, they write a brief message that describes the purpose of the change, including which files are being modified and why. Change log messages can be analyzed to characterize the type of change that was made, such as a bug-fix change.

Revision (Transaction): Groups of file changes in one commit are called a revision or transaction. Usually revisions are in chronological order. For example, revision 1 is a prior change group to revision 2. Some SCM systems, such as CVS, use the term 'revision' to represent an individual file change. For example, in CVS each changed file in one transaction has different revisions. The term 'revision' is used to represent a group of changes in one commit in other SCM systems such as

Subversion. In this dissertation, a revision or a transaction is a group of file changes in one commit.

Bug (defect): A bug is zero or more lines of source code whose inclusion (or omission) causes anomalous software behavior. The term defect and fault are also widely used to describe anomalous software behavior [20, 66]. The term bug has slightly different meanings when it is used in different domains or used by different researchers. In this dissertation, the term bug describes the actual source lines that cause anomalous software behavior. The following sentence shows the context and meaning of the term bug used in this dissertation:

"A **bug** in line #39 in *Foo.c* causes this application to hang."

Fix: When developers locate bugs, they are required to repair the bug to remove the anomalous software behavior. Developers remove buggy source code and replace it with correct code. The act of removing bug(s) is called a fix, typically enacted by changing buggy source code.

Fix change: A change that includes more than one fix is called a fix change. Since a fix change removes buggy code and replaces it with correct code, the removed code is considered to be buggy code.

Bug-introducing change (buggy change): The change that initially introduces a bug is called a bug-introducing change.

Change delta: In a file change, the changed lines are called the change delta. The deleted lines in the old file are called the deleted delta, and added lines in the new file are called the added delta.

Hunk: In a file change, contiguous line changes in a single file are called a hunk, as shown in Figure 2-3. A file change delta may include more than one hunk, since the changed lines in a delta may be sparse.

Bug hunk: If a change is a fix, the deleted or modified part of source code in the old file is considered buggy and is called a bug hunk.

Fix hunk: If a change is a fix, the added or modified part of source code in the new file is considered to be a fix and is called a fix hunk.

Feature (factor): In this dissertation, a feature means a property of a software change and is used for classifying changes. For example, the author, commit date, and a keyword in the deleted delta are features of changes. In the software engineering literature, the term feature usually means a distinct software functionality, but in the machine learning literature, a feature is a factor or properties of instances in a training or testing corpus [2].

Feature engineering: A collection of techniques to select good (predictive) features and extract them from change instances.

Corpus: The collection of data used to train and test prediction models. In this dissertation, a corpus consists of a series of instances, where each instance corresponds to a project revision. Each instance has an associated set of features.

3 Making a Corpus from Software History

The proposed bug prediction models in this dissertation are based on software history mining that involves the extraction of useful information from software evolution data as stored in a project's SCM repository. The collection of extracted data is called a corpus following machine learning naming conventions. The corpus is also used to evaluate the bug prediction models. To extract raw data from SCM systems such as CVS [4] or Subversion (SVN) [3], the extractor must understand the system's data model and interface.

The Kenyon infrastructure [6] and APFEL [12] (Columba, Eclipse, and Mozilla for the bug cache) is used to extract raw data from SCM systems including change instances, source code, change logs, authors, and change times. Kenyon is a data extraction, preprocessing, and storage backend designed to facilitate software evolution research. Kenyon was developed at UC Santa Cruz by Jennifer Bevan.

Processing the raw data falls into two folds: labeling change instances and extracting useful features from extracted instances. The labeling process identifies each change instance as a fix, bug-introducing, or normal change. The feature extracting process tries to determine useful patterns in fix or bug-introducing changes, so the patterns can be applied for bug prediction.

This chapter explains the process of data extraction from an SCM repository and making a corpus from the extracted data.

3.1 Systems Analyzed

For this dissertation, 12 open source projects, listed in Table 3-1, are used. These projects are chosen due to ease of access to the full source code, bug reports, and entire project history.

Table 3-1. 12 open source projects analyzed.

Project	Language	Software type	SCM
Apache HTTP 1.3 (A1)	C	HTTP server	SVN
Bugzilla (BUG)	Perl	Bug tracker	CVS
Columba (COL)	Java	Mail client	CVS
Gaim (GAI)	C/C++	Instant messenger	CVS
GForge (GFO)	PHP	Collaborative development	CVS
Jedit (JED)	Java	Editor	CVS
Mozilla (MOZ)	C/C++/JS/XML	Web browser	CVS
Eclipse JDT (ECL)	Java	Java development/IDE	CVS
Plone (PLO)	Python	Content management	SVN
PostgreSQL (POS)	C/C++	DBMS	CVS
Scarab (SCA)	Java	Bug tracker	SVN
Subversion (SVN)	C	SCM	SVN

A more detailed description of each project is described as follows:

Apache HTTP 1.3 (http://httpd.apache.org/) – Apache HTTP 1.3 is the most widely used HTTP server, with more than 70% of the web sites on the Internet using Apache [48] in 2006.

Bugzilla (http://www.bugzilla.org/) – Bugzilla is a bug tracking system used by the Mozilla project and other open source and commercial projects.

Columba (http://www.columbamail.org/drupal/) – Columba is a GUI email client.

Gaim (http://gaim.sourceforge.net/) – Gaim is a multi-protocol instant massaging system that supports multiple instant messaging protocols including Microsoft Network Messengers, Yahoo Messenger, Jabber, AOL Instant Messenger.

GForge (http://gforge.org/) – GForge is a collaborative development environment for software development. Developers can publish their own project homepage, mailing list, discussion forum, and issue tracking system using the GForge software.

Jedit (http://www.jedit.org/) – Jedit is a programmer's text editor that supports plug-ins. It is a highly configurable and customizable editor.

Mozilla (http://www.mozilla.org/) – Mozilla is a Netscape-based open source web browser.

Eclipse JDT (http://www.eclipse.org/) – Eclipse is a universal and extendable integrated development environment (IDE) for software development. JDT is a part of Eclipse project that provides Java Development Tools.

Plone (http://plone.org/) – Plone is an open source content management system that helps non-technical users to add, edit, and manage web pages.

PostgreSQL (http://www.postgresql.org/) – PostgreSQL is a widely used database system.

Scarab (http://scarab.tigris.org/) – Scarab is a Bugzilla-like bug tracking system that is highly customizable.

Subversion (http://subversion.tigris.org/) – A source code version control system similar to CVS.

3.2 Change History Extraction and Transaction Recovery

Kenyon automatically checks out the source code of each revision and extracts change information such as the change log, author, change date, source code, change delta, and change metadata. This data is then fed into the feature generation process to convert a file change into patterns.

A commit can include more than one file change and it is important to know what files are changed together in one commit. For example, if you want to know how many files are changed in a commit on average, you should know the number of file changes in every commit. Information on changed files in a commit, also called co-changed files, is widely used in software evolution research [5, 7, 17, 68].

Some SCM systems, such as Subversion, keep co-change file information for every commit. Unfortunately other SCM systems, such as CVS, do not keep this transaction information. CVS only provides versioning at the file level, and does not record co-change information between files. In this case, a transaction recovery process is necessary before any other analysis of software evolution can take place.

The basic idea of transaction recovery is that co-changed files are committed over a short time span, such as 200 seconds, and have the same author and change log message. To recover per-product transactions from CVS archives, individual per-file changes are grouped into an individual transaction using the sliding window approach introduced in [69]. Two subsequent changes d_i and d_i+1 by the same author and with the same log message are part of one transaction if they are at most 200 seconds apart. This transaction recovery algorithm is implemented in Kenyon [6].

3.3 Identifying Fix Changes

Fix identification is the first step towards finding out when bugs are introduced. There are some heuristic ways to identify fix changes in software history that rely on the change logs left by the developers [10, 13, 58]. If the change logs can provide any clue that indicates that this revision is fixing some problems, it is then considered a fix revision. Two fix identification algorithms are used for this step: searching for special keywords that indicate fixes, such as "Fixed" or "Bug" [42], and searching for references to bug reports like “#42233” [10, 13, 58]. This heuristic identifies whether an entire revision contains a bug-fix. If it does, all files in the revision are marked as fix changes. Manual inspection of the change logs for each project is used to identify the keywords that indicate fix revisions for each project. The project keywords are shown in Table 3-2.

Table 3-2. Keywords and reference identifiers used to find fix revisions. * bug id reference is a 7-digit number.

Project	Keywords or Phrases
Apache HTTP 1.3 (A1)	Fix, bug, error
Bugzilla (BUG)	Fix, bug, * bug id reference number
Columba (COL)	[bug], [bugfix]
Gaim (GAI)	Patch, fix, bug
GForge (GFO)	Patch, fix, bug
Jedit (JED)	Patch, fix, bug
Mozilla (MOZ)	* bug id reference number
Eclipse JDT (ECL)	* bug id reference number
Plone (PLO)	Patch, fix, bug
PostgreSQL (POS)	Patch, fix, bug
Scarab (SCA)	Patch, fix, bug, issue number
Subversion (SVN)	Fixed issue number

3.4 Identifying Bug-introducing Changes

The bug-introducing change identification algorithm proposed by Śliwerski, Zimmermann, and Zeller (SZZ algorithm) [58] is used in this dissertation. After identifying bug-fixes, SZZ runs a *diff* tool to determine what changed in the bug-fixes. The diff tool returns a list of regions that differ in the two files; each region is called a hunk. SZZ 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 *annotate* feature of SCM systems. The annotate feature computes, for each line in the source code, the most recent revision in which the line was changed, and the developer who made the change. The discovered origins are identified as bug-introducing changes.

Revision 1 (by kim, bug-introducing)

```
1 kim | 1: public void bar() {
1 kim | 2:   // print report
1 kim | 3:   if (report == null) {
1 kim | 4:       println(report);
1 kim | 5:   }
```

Revision 2 (by ejw)

```
2 ejw | 1: public void foo() {
1 kim | 2:   // print report
1 kim | 3:   if (report == null){
2 ejw | 4:       println(report.str);
1 kim | 5:   }
```

Revision 3 (by kai, bug-fix)

```
2 ejw | 1: public void foo() {
1 kim | 2:   // print report
3 kai | 3:   if (report != null) {
1 kim | 4:       println(report);
1 kim | 5:   }
```

Figure 3-1. Example bug-fix and source code changes. A null-value checking bug is injected in revision 1, and fixed in revision 3.

Figure 3-1 shows an example of the history of development of a single function over three revisions:

Revision 1 shows the initial creation of function *bar*, and the injection of a bug into the software, the line *‘if (report == null) {‘* which should be *‘!=’* instead. The leftmost column of each revision shows the output of the SCM annotate command, identifying the most recent revision and the developer who made the revision. Since this is the first revision, all lines were first modified at revision 1 by the initial developer ‘*kim.’* The second column of numbers in revision 1 lists line numbers within that revision.

In the second revision, two changes were made. The function *bar* was renamed to *foo*, and *println* has argument ‘*report.str*’ instead of ‘*report.*’ As a result, the annotate output shows lines 1 and 4 as having been most recently modified in revision 2 by *‘ejw.’*

Revision 3 shows a change, the actual bug-fix, changing line 3 from ‘*==’* to *‘!=’*.

The SZZ algorithm tries to identify the bug-introducing change associated with the bug-fix in revision 3. SZZ starts by computing the delta between revisions 3 and 2, yielding the line 3. SZZ then uses SCM annotate data to determine the initial origin of line 3 at revision 2. Finally SZZ identifies the bug-introducing change, revision 1, by looking at the annotation for line 3 at revision 2.

There are limitations of SZZ, such as false positive identification for commenting and formatting changes. These limitations are addressed by ignoring comment changes and source code reformatting changes [32]. However, the original SZZ [58] is used in this dissertation.

4 Bug Cache

Software quality assurance is inherently a resource-constrained activity. In the vast majority of software projects, the time and people available for verification and validation are not sufficient to eliminate all bugs before the software is released. Any technique that allows software quality engineers to reliably identify the most bug prone software functions provides multiple benefits. It allows available verification and validation resources to be focused on the functions that have the most bugs. If developers have a potential buggy function list, it is possible to selectively use time intensive techniques such as software inspections, formal methods, and various kinds of static code analysis. It also provides an understanding of which functions are likely to be troublesome once a system has been released, thereby allowing customer service organizations to plan ahead for their presence.

The software bug cache algorithm is executed over the revision history of a software project, yielding a small subset (usually 10%) of the project's files or functions/methods that are the most bug prone. This bug cache has many uses. A developer can run through the contained locations, performing a detailed examination of each function, and eliminate all bugs that he or she may encounter.

He or she can convene a code inspection team, and analyze the code from multiple perspectives. In addition, she could perform one or more static code analyses, knowing that the time spent wading through false positive results is worthwhile, since she will also find bugs. A tester can use the bug cache to develop new test cases for bug-prone locations. He can also examine the list to determine classes of frequently occurring bugs, tailoring the test cases to catch recurring bug types.

Two important qualities of software bug prediction algorithms are accuracy and granularity. The accuracy is the degree to which the algorithm correctly identifies future bugs. The granularity specifies the locality of the prediction. Typical bug prediction granularities are the executable binary [46], a module (often a directory of source code) [23], or a source code file [50]. For example, a directory level of granularity means that predictions will indicate that a bug will occur somewhere within a directory of source code. The smaller the granularity, the more precise a prediction algorithm must be to achieve a given level of accuracy. The most difficult granularity for prediction is the entity level (or below), where an “entity” is a function or method, depending on whether the software is procedural or object-oriented.

This chapter presents a bug cache algorithm that is 73%-95% accurate at predicting future bugs at the file level of granularity, and 49%-68% accurate at the entity level. This accuracy is better than or equivalent to other efforts reported in the literature. Moreover, the bug cache is able to achieve this predictive accuracy at the entity and file level of granularity, which permits better allocation of testing resources due to the greater precision of the prediction.

The key insight that drives the bug prediction algorithm is the recognition that most bugs are local. Bugs do not occur uniformly in time across the history of a function; they are bursty. The hypothesis is that there are bug bursts if a function has recently introduced a bug, been changed, been created, or is in near proximity to other buggy functions. Specifically, bug occurrences have four different kinds of locality:

Temporal Locality. If an entity introduced a bug recently, it will tend to introduce other bugs soon. This is similar to the “most recently fixed” heuristic in [23].

Changed-entity Locality. If an entity has been changed recently, it will tend to introduce new bugs soon. This is equivalent to the notion of code churn in [46] and the “most recently modified” heuristic in [23].

New-entity Locality. If an entity has been added recently, it will tend to introduce new bugs soon.

Spatial Locality. If an entity introduced a bug recently, “nearby” entities (in the sense of logical coupling [17] described in Section 4.1.4) will introduce bugs soon.

Like Hassan and Holt [23], the notion of a cache was borrowed from computer and operating systems research, and applied here for the purpose of bug prediction. However, the algorithm does not use the bug cache in a typical manner, to ensure fast access to frequently used data. Instead the algorithm uses the cache as a convenient mechanism for holding the current list of the most bug prone entities, and for aggregating together multiple heuristics for adding and removing entities from the cache. The switch to using a cache involves a subtle, but important shift in approach in which there is no need to create massive mathematical equations to predict future bugs. Instead the algorithm performs cache selection and removal based on entities that meet specific criteria. The research goal of the bug cache is to develop algorithms that minimize the size of the bug cache while maximizing its predictive accuracy.

Unlike most existing research on bug prediction, in this study bug-introducing changes are used as bug datasets, instead of fixes. A fix tells us where a bug occurred (which lines and files), but it does not say when the bug was added. However, four of the bug localities depend on accurately identifying when a bug occurred, especially for spatial locality. Correct identification of when a bug occurred makes it possible to identify nearby logically coupled software *at that time*. A bug-introducing change is the software modification(s) that introduced a bug that was later fixed. It tells us when the bug occurred, and hence the chronology of events is bug-introducing change(s), bug report, and then fix.

This chapter describes algorithms for adding and removing entities from the bug cache based on the bug localities, and evaluating these algorithms. The basic process is as follows:

Initialization:

1. Extract fixes from a project by mining its software history as stored in an SCM repository. The techniques described in Chapter 3 are used for this step.

2. Extract bug-introducing changes at the entity level by analyzing the revision history to identify changes to the entity that precede the fix and bug report, as described in Chapter 3.

3. Pre-load the bug cache with the largest entities (by lines of code, or LOC) in the initial project revision, creating the initial bug cache state. The reason for doing this is the common understanding that long LOC entities are more bug prone.

Cache operation:

4. Observe the locations of bug-introducing changes at revision *n*. If a bug occurs in an entity that is currently in the bug cache, it is a cache hit. Otherwise it is a miss. A cache hit can be viewed as a successful prediction of the bug. If the algorithm misses a bug, the algorithm fetches the entity (temporal locality) and nearby entities (spatial locality) and brings them into the cache for use in future bug predictions starting at revision *n+1*.

5. At revision *n*, the algorithm pre-fetches entities that have been created (new entity locality) and modified (changed-entity locality) between revisions *n-1* and *n*.
6. Since the size of the bug cache is fixed, when the algorithm adds an entity, it must remove an entity. The algorithm applies a cache replacement policy, such as least recently used, to decide which entities should be replaced.
7. Iterate over steps 4-6 to cover the existing revision history and bug information.

After following the process, the hit rate is computed by:

$$hit\, rate = \frac{\#of\ hit}{\#of\ hit + \#of\ miss}$$

If the hit rate is close to 1, it means that most future bugs occur in the entities contained in the bug cache.

Previous work on bug detection and prediction falls into one of the following categories: identifying a problematic module list by applying software quality metrics [23, 27, 28, 50, 51], predicting the bug density of each module by analyzing its software change history [20, 46], and detecting bugs by analyzing source code using type checking, deadlock detection, or pattern reorganization [14, 38, 60]. This related work is discussed in detail in Chapter 7. The bug cache approach differs from this previous work in that I:

Use bug-introducing changes: Most previous research uses bug-fix data to predict or validate their predictions. This is a potential source of error since a fix gives *where* but not *when* information. Previous research was unable to explore temporal bug locality since fixes cannot indicate the time a bug was first injected into the software. Since bug-introducing changes are used, it is possible to know when bugs occurred, and to evaluate temporal bug locality.

Use the spatial distance (distance among functions using logical coupling in co-changes [17], described in Section 4.1.4) of functions as one of the predictors. Analysis of the Apache 1.3 project shows that this predictor alone accounts for 18% of the entities in the cache.

Aggregate multiple selection algorithms via the cache: The use of a cache permits the simultaneous application of several heuristics, and separates the concerns of selecting and deselecting bug-prone entities. In addition, the bug cache permits selection algorithms to be adaptive to changing conditions.

Focus on small prediction granularity, at both the entity and file level. This level of granularity is more useful in practice since it permits a more focused application of software quality resources.

Use an on-line learning approach: The bug cache approach is similar to on-line machine learning algorithms [2]. A prediction model (bug cache) is trained using *1* to *n-1* revision data to predict bugs at revision *n*. Based on the prediction result, the bug cache approach modifies the prediction model to predict the next bug.

The remainder of this chapter starts by providing a discussion of the four kinds of bug localities (Chapter 4.1) and bug cache operations that include pre-fetching and cache replacement strategies (Chapter 4.2). Following are results from operating the bug cache on seven projects at the file and entity level (Chapter 4.3), along with an analysis of cache replacement policies and sensitivity analysis of the four bug localities. Chapter 4.4 provides discussion of these results, and Chapter 4.5 summarizes the bug cache approach.

4.1 Bug Localities

Software engineering does not yet have a solid model for why programmers create software bugs. Programmers do not intentionally set out to create bugs. It is likely that some kind of programmer cognitive error causes bugs, such as poor understanding of the software being modified, incorrect model of the operational environment, or poor comprehension of the requirements. A programmer operating with incorrect models or understanding will likely have multiple impacts on the code as he makes modifications since his incorrect background information will lead him to inject multiple bugs. Hence, when a software bug is found, it is viewed as an indication of programmer cognition error, and that this error likely resulted in multiple software bugs. When a bug is found, the entity harboring the bug is assumed to have additional bugs that will soon be observed (*temporal locality*), and other entities modified at the same time are strongly suspected (*spatial locality*). Code modification is risky since the programmer must have a correct understanding of the code, requirements, and details of the modification request, and hence all additions or modifications of entities are also potential bug locations (changed entity and new entity localities). The temporal and spatial localities are described in more detail below.

4.1.1 Temporal Locality

The intuition behind temporal locality is that a software entity with an observed bug will soon have more bugs. In a similar vein, an entity that has recently been changed will soon have bugs, a correlation noted by several researchers [20, 43]. The time when a bug was added into the code (i.e. when the bug-introducing change was made) is an important factor that correlates to future bug density. If a bug occurred several years ago, and was fixed one year ago, there is less of a chance that the old bug can cause a new bug to form. But if a bug was introduced a week ago and fixed two days ago, it is much more likely that the bug is clustered with other bugs that will soon be observed.

Temporal locality is different from using the accumulated numbers of changes or fixes to predict future bugs. For example, assume there are two software entities with observed bugs (circles) as shown in Figure 4-1. Entity A has three bugs, but they are relatively old. Entity B only has two bugs, but they are relatively new. If the total number of bugs is used as a future bug predictor, A is more likely to be predicted as buggy rather than B. However, since entity A's bugs are older, it is more likely that the manifestations of the original programmer's error have already been found. Hence, even though entity A has more bugs in the past, entity B has a greater likelihood of bugs in the near future since entity B's bugs are fresher.

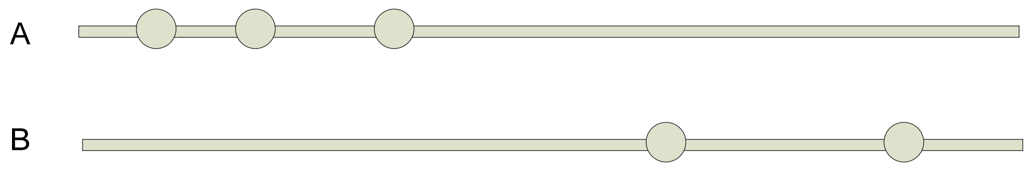

Figure 4-1. Observed bugs in two hypothetical software entities. Time progresses from left to right.

Graves's weighted time damp model is similar in spirit to temporal locality. It more heavily weights recent bugs to predict future ones, and this model was the best one they observed [20]. The bug cache with temporal locality has the advantage of a more simple description than the math-heavy weighted time damp model.

Temporal locality also guides cache removal strategies. If there are no bugs in an entity for a long time, the bug cache removes the entity from the cache, and assumes there are no further bugs.

4.1.2 Changed Entity Locality

There is a common understanding that if an entity changes often, it may be instable and hence introduce bugs [5, 17]. If an entity has been changed recently, the bug cache pre-fetches the changed entity into the cache assuming that it will tend to introduce new bugs soon. This is equivalent to the notion of code churn in [46] and the "most recently modified" heuristic in [23].

4.1.3 New Entity Locality

Similar to the changed entity locality, if an entity has been added recently, the bug cache pre-fetches the added entity into the bug cache assuming it will tend to introduce new bugs soon.

4.1.4 Spatial Locality

If a software entity has a bug, there is a good chance that other nearby entities may also have bugs that will soon be observed.

What are nearby entities? There are several ways to define distance in software. One way is using physical distances among entities. For example, the entities in the same file or directory could be nearby entities. Another way is using logical coupling among software entities [5, 17].

Co-change data is used to compute logical coupling between entities. If two entities are modified together many times, they have a short distance to each other.

This reflects that they are logically "close" to one another. The distance between any two entities is computed using the modified graph data structure from CCVisu [7]. CCVisu is a tool that extracts a co-change graph from a software repository.

The bug cache algorithm starts by creating a graph structure among the entities changed at a given revision *n*, creating a node for each entity, and for the revision itself. The algorithm creates arcs between the revision node and each entity node. As an example, consider revision 1 (R_1) where entities E_1 and E_2 are changed together, and revision 2 (R_2) where E_1, E_2 and E_3 changed together. Figure 4-2 (a) and (b) shows the change graphs for the two revisions.

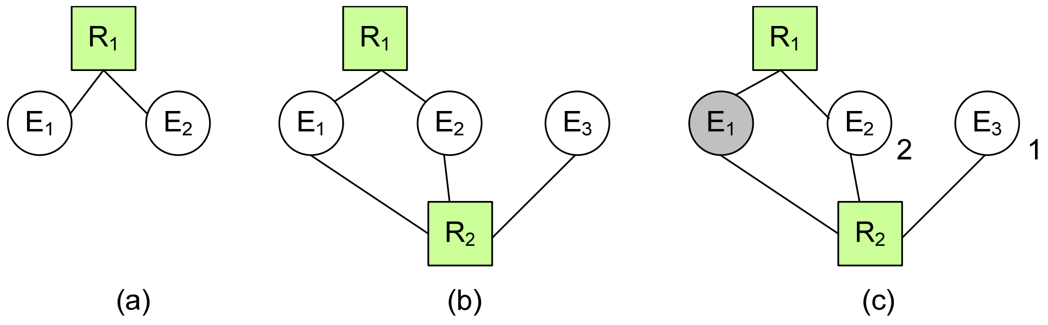

Figure 4-2. Change graphs and distance between two entities. Circles represent entities, and rectangles revisions.

Given the two change graphs shown in Figure 4-2 (b), the distance is computed as follows:

1. Select any two nodes to compute the distance.

2. Count the number of the direct revision paths between the two nodes. A direct revision path between two nodes, *u* and *v*, is a unique path *u*-*r*-*v*, where *r* is any revision node.

The numbers in Figure 4-2 (c) show the number of direct revision paths from E_1 to the corresponding nodes, indicating the distance between them. A high number means two nodes are "close" (logically coupled), and so E_1 and E_2 are closer than E_1 and E_3.

When there is a bug in an entity, the bug cache loads the nearby entities using the number of the direct revision paths between entity nodes. The notion of *block size* was adapted from existing cache work to describe how many entities the bug cache loads into the cache. A block size of *b* indicates that the bug cache approach loads the top *b-1* closest entities (i.e. the ones with the highest path counts) into the cache, along with the buggy entity itself. In the analysis of bug caching, the effects of different block size values were analyzed.

4.2 Operation of the Bug Cache

This chapter describes the operation of the bug cache in detail.

4.2.1 Basic Operation

The bug cache contains a subset of a software project's files or entities. The bug cache approach places no constraints on the size of the cache, and hence it can be adjusted based on the resources available to perform intensive verification of cache entries. This analysis typically uses a cache size of 10% of the total number of files or entities, since it provides a reasonable tradeoff between the cache size and resulting accuracy. Larger cache sizes result in greater hit rates (better accuracy), but with the bugs spread out over a greater number of files or entities (lower precision).

The bug cache maintains a list of what the bug cache algorithm has chosen as the most bug prone software entities. For every revision, the bug cache approach examines the list of bugs found there. If a new bug is found in an entity that is currently in the bug cache, it is a hit, and hence it has correctly predicted that the entity was bug prone. If a bug is not found, it is a miss. In the case of a miss, the bug cache algorithm fetches the missed entity into the cache (temporal locality) and fetches other entities (spatial locality), depending on the block size and whether any other entities are nearby. This approach is similar to on-line machine learning algorithms [2] in that the bug cache learns from the bug distributions (hits and misses) and quickly updates its prediction model (list in the cache).

4.2.2 Pre-fetches

Multiple pre-fetching techniques are used to improve the hit rate of the bug cache. The motivation of pre-fetching is as follows. Assume bug cache does no pre-fetching and only loaded entities when a bug was encountered. If there are *n* entities that contain at least one bug, by not pre-fetching, it is guaranteed that there will be more than *n* misses since an entity is only fetched when a bug was found in the entity. The entity with the first bug will be automatically missed – compulsory miss. To reduce the compulsory miss count, pre-fetching potential buggy entities in advance is necessary. There are three kinds of pre-fetch algorithms, described below.

Initial pre-fetch: Initially the bug cache is totally empty, and in the absence of pre-fetching, this would lead to many cache misses. To reduce initial-miss, the bug cache approach initializes the cache with entities that are likely to have bugs. Since at this moment any history information is unavailable, LOC is used as the initial predictor since it is one of the best bug predictors in the literature [20, 50].

New entity pre-fetch: Between two project revisions it is common that new entities are added, and some are deleted. If an entity has been deleted, there is no chance it can have future bugs, so the bug cache approach unloads the deleted entity from the cache, if present. When there are new entities, those new entities having the largest LOC count (new-entity locality) are pre-fetched. The new entity pre-fetch size is controlled by a bug cache option.

Changed entity pre-fetch: Previous research has used the number of changes as a dominant bug predictor. This is incorporated into the bug cache as the notion of changed-entity locality. If an entity has been changed, the entity is pre-fetched into the cache. The maximum number of changed entity pre-fetches per revision is controlled by an option.

4.2.3 Cache Replacement Policies

Since the bug cache has limited size, to fetch more software entities into the cache, the bug cache approach must unload some entities first. Which entity should be unloaded and which one has the least potential for new bugs?

In memory cache systems, the LRU (Least Recently Used) algorithm replaces the data that was used the longest time ago. It is in common use due to its simplicity and strong performance. Similarly, this study explores LRU-like algorithms to decide which entity should be unloaded.

Previous research has shown that entities with many changes or prior bugs are likely to have additional bugs [20, 23, 50]. Based on these results, this study also explores change-weighted and previous-bug-weighted LRU algorithms.

Table 4-1. Sample cached entities and their properties.

Id	Last found bug/hit (ago)	Cumulative changes	Cumulative bugs
1	1 day	8	1
2	10 days	7	5
3	9 days	6	7
4	2 days	5	4

Least Recently Used (LRU). This algorithm unloads the entity that has the least recently found bug (hit). Consider a cache with the entities shown in Table 4-1. Based on the LRU algorithm, entity 2 will be unloaded, since it is the least recently used entity.

Change count weighted (CHANGE). Previous research used the accumulated number of changes as a future bug predictor [20] based on the idea that if an entity has been changed many times in the past, it is more likely to have bugs in the future. Such entities should be kept in the bug cache as long as possible. The bug cache approach compares the accumulated change numbers of each entity in the cache, and unloads the entity that has the least amount of change. According to this policy, entity 4 in Table 4-1 will be unloaded since it has never been changed.

Bug count weighted LRU (BUG). This policy is similar to the change count weighted LRU. It uses the accumulated count of observed bugs to decide which entity should be unloaded, removing the one with the fewest total bugs. The intuition here is if an entity has had many bugs, it will likely continue to have bugs.

Following this policy, entity 1 in Table 4-1 will be unloaded since it has the fewest number of bugs.

4.2.4 Cache Operation

Bug cache operation involves the combined operation of fetches into the cache and a replacement policy. This section provides an extended example to illustrate cache operation.

Suppose there is a software project with three software entities, 1, 2, and 3 (as shown in Figure 4-3). A new entity 4 is added at revision 4. Assume that the bug cache size is 2 (this is 50% of the size of the hypothetical project in Figure 4-3). Initially the bug cache is empty. The initial pre-fetch process loads entities 1 and 2 into the cache (assuming entities 1 and 2 are the largest entities). Revision 1 has no bugs, and hence does not affect the cache.

At revision 2, entities 1 and 2 are changed and entity 2 has a bug. There is a bug cache hit for entity 2 since it has a bug. Since entity 1 is changed, it needs to be loaded into the cache. However, it is already in the cache, so there is no cache change. The reason for fetching entity 1 is changed from initial pre-fetch (i) to changed entity pre-fetch (c).

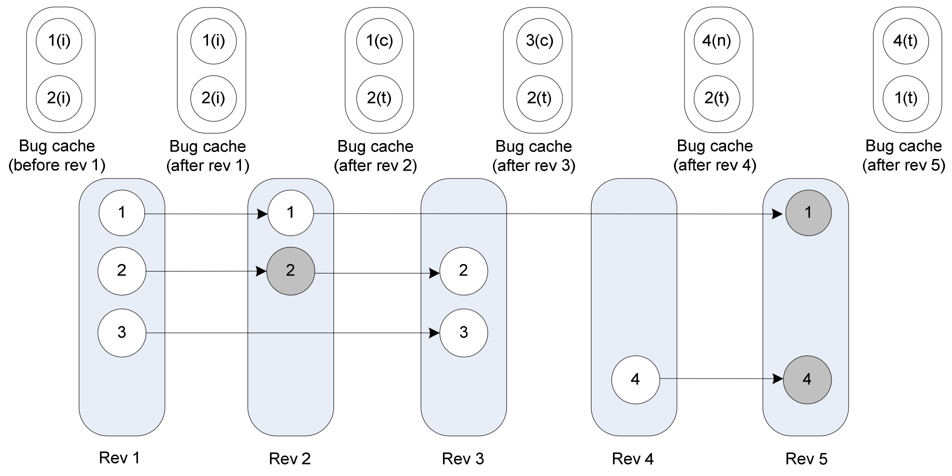

Figure 4-3. Bug cache operation example. The grayed entity indicates a bug entity. The bug cache size is 2. Each entity in the bug cache has the reason of each fetched entity: (i) for initial fetch, (t) for temporal locality, (c) for changed entity, (s) for spatial locality, and (n) for new entity.

At revision 3, entities 2 and 3 are changed. The bug cache approach tries to fetch these changed entities, but the cache is full and some entities need to be replaced. Using the LRU cache replacement approach entity 1 is replaced since entity 2 has one hit and entity 1 has zero hits. After revision 3, entities 2 and 3 are in the bug cache.

At revision 4, new entity 4 is added into the software. After revision 4, the new entity 4 will be fetched into the bug cache by replacing entity 3 in the cache, since the hit count of entity 3 (zero) is no greater than the hit count of entity 4 (also zero), and entity 4 has been changed more recently.

Finally, at revision 5, entities 1 and 4 are changed, and entities 1 and 4 have a bug. For entity 4, there is a bug cache hit and the entity remains in the bug cache. For entity 1, there is a cache miss, so the bug cache attempts to fetch entity 1. By applying LRU, entity 2 is replaced by entity 1. The algorithm then tries to find the entities that co-changed with entity 1. Entity 2 was changed with entity 1 at revision 2, so entity 2 has co-changed with entity 1. The bug cache tries to load entity 2 (spatial locality), but entities 1 and 4 are least recently hit entities. The cache remains as-is. After revision 5, entities 2 and 4 are in the bug cache and are used for bug prediction in the following revision.

4.3 Case Study

The bug caching algorithm has multiple parameters that can be modified, all of which affect the hit rate of the algorithm. It is possible to modify the cache size, block size, pre-fetch size, and cache replacement policy. To determine which combination of parameters yields the best hit rate, a brute force cache analysis that iterated through multiple option combinations was performed. The performance of different cache replacement strategies was also evaluated, along with the relative contributions of each kind of bug locality, the impact of different granularities, and the impact of improved techniques for finding bug-introducing changes. Table 4-2 lists the analyzed projects for the bug cache case study.

Table 4-2. Analyzed open source projects for the bug cache. The period shows the analyzed project timespan. The number of revisions indicates the number of revisions extracted. The number of entities indicates number of functions or methods in the last revision. The number of bugs indicates the number of bug-introducing changes extracted by mining the change logs and change histories of each project. For the Eclipse project only the core.jdt module is used due to the large size of the entire project. Similarly, only the Mozilla/content/ modules are used for the Mozilla project.

Project	Period	# of revisions	# of entities	# of files	# of bugs
A1	01/1996 ~ 07/2005	7,747	2,113	154	1,954
SVN	08/2001 ~ 07/2005	6,029	3,693	255	1,566
POS	04/1996 ~ 08/2005	14,650	8659	598	19,902
MOZ	03/1998 ~ 01/2005	109,636	8203	396	52,265
JED	09/2001 ~ 06/2005	1,386	5429	420	3,060
COL	11/2002 ~ 07/2005	2,848	8428	1428	720
ECL	04/2001 ~ 01/ 2005	78,948	33214	3330	15,217

4.3.1 Hit Rates

To compute hit rate, cache options are set up as constants, such as a cache size of 10%, block size of 5%, and pre-fetch size of 1% of the total number of functions/ methods. For example, Subversion has 3,693 functions. The cache size for Subversion is 369, block size is 184, and pre-fetch size is 36. The hit rates of file and function/method levels with three cache replacement algorithms are shown in Figure 4-4 and Figure 4-5. Numbers in parenthesis in Figure 4-4 and Figure 4-5 indicate cache size, block size and change/new entity pre-fetch size. File level hit rates are 57%-93%, and entity level hit rates are 28%-68% depending on the cache replacement algorithm.

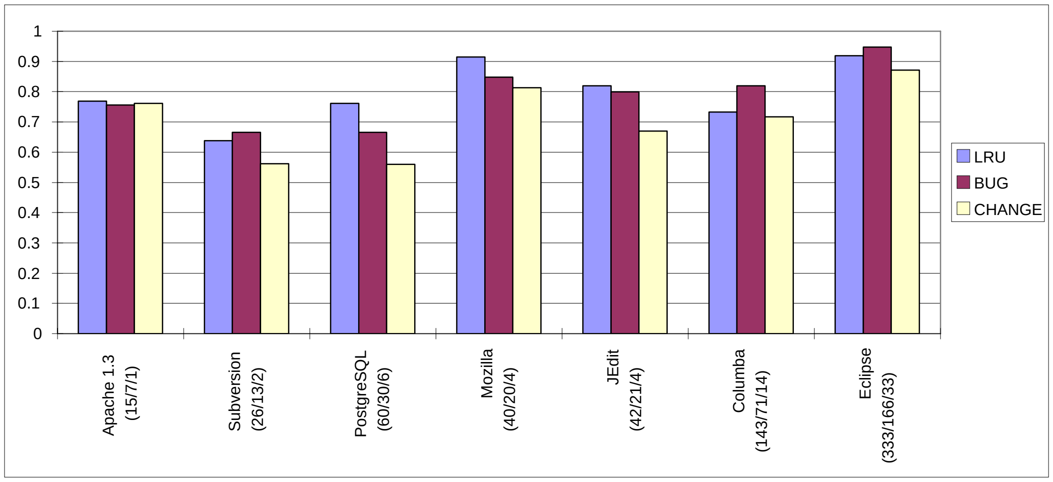

Figure 4-4. File level hit rates using different cache replacement algorithms. The cache size is 10%, block size is 5%, and pre-fetch size is 1% of the total number of files.

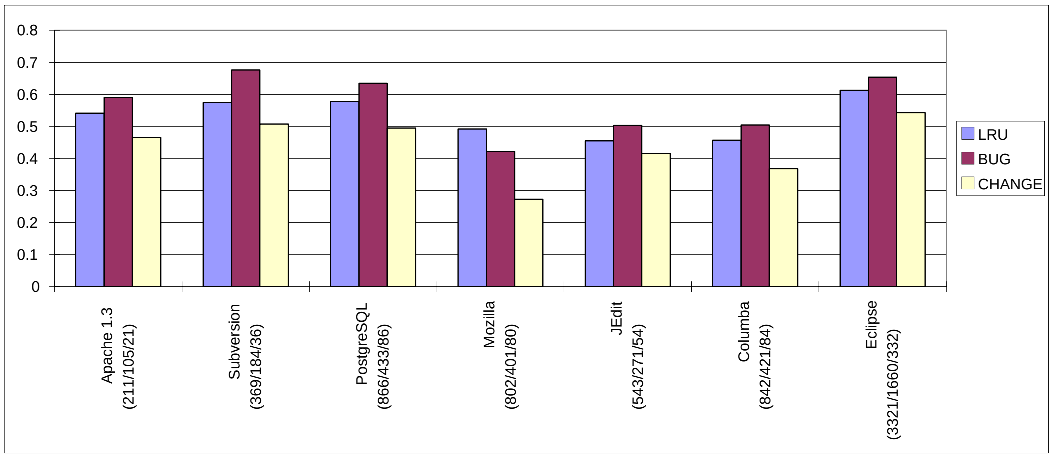

Figure 4-5. Function/method level hit rates using different cache replacement algorithms. The cache size is 10%, block size is 5%, and pre-fetch size is 1% of the total number of functions/ methods.

Using the same cache size with different sets of cache options, such as cache replacement algorithms and block sizes, yields varying hit rates. For the brute force analysis of different cache option combinations, the cache size is fixed at 10% of the total number of entities or files. Other options such as the block size, pre-fetch sizes, and cache replacement algorithms are changed, and the hit rate results were observed. Block sizes were varied from 0 to 100% of the cache size with a step of 5%. Similarly pre-fetch sizes were varied from 0 to 100% of the cache size (step 5%) to determine the best hit rate of each project. An analysis was performed at two software granularity levels: file and entity (function/method).

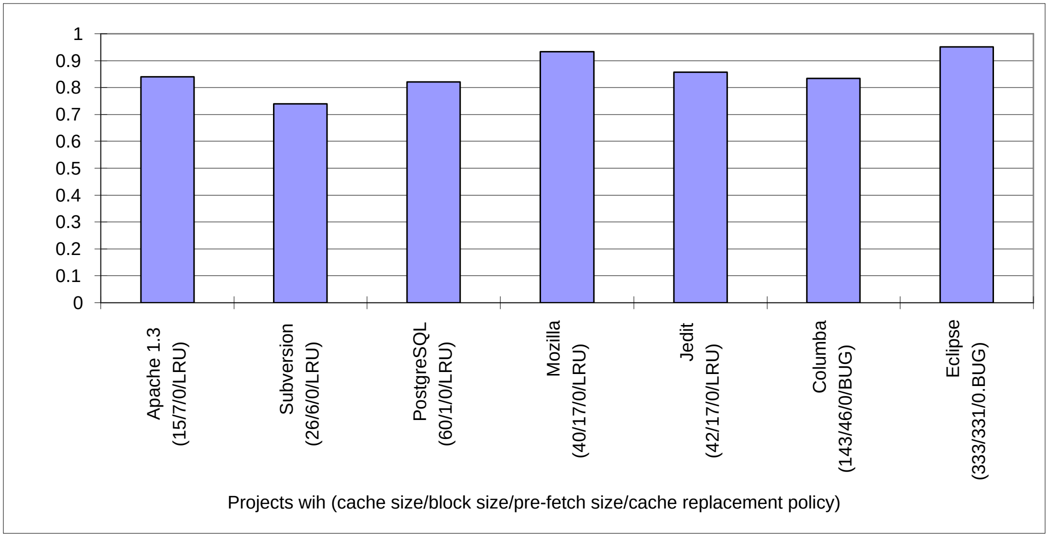

Figure 4-6. Hit rate of the file level cache, with cache size set to 10% of all project files.

The best option combinations for each project and the resulting hit rates are shown in Figure 4-6 (file level) and Figure 4-7 (entity level). To summarize, the best combination of bug cache parameters yields hit rates (predictive accuracy) of 73-95% at the file level, with typical performance in the low to mid 80s. The most directly comparable work is [23], which also uses a caching approach, but at the module (directory) level of granularity. The best hit rates in [23] vary from 32% to 90% for a cache size of 10 modules with typical performance in the upper 40s (the

90% hit rate was for only one system, and only was present at very early revisions; steady state behavior for the same system was in the mid 50s.) For a cache size of 10% of all modules, hit rates in [23] vary from 45%-82% using their best performing heuristic, most-frequently-modified.

Ostrand et al. [50] predicted the bug density of each file in a software project using a negative binomial linear regression over various factors. Based on the predicted bug density, they ordered the files from most to least bug dense. Using this method and selecting the top 20% of the files, they predict 71-93% of future bugs. The bug cache approach achieves similar accuracy, but with only 10% of project files, which is twice the precision.

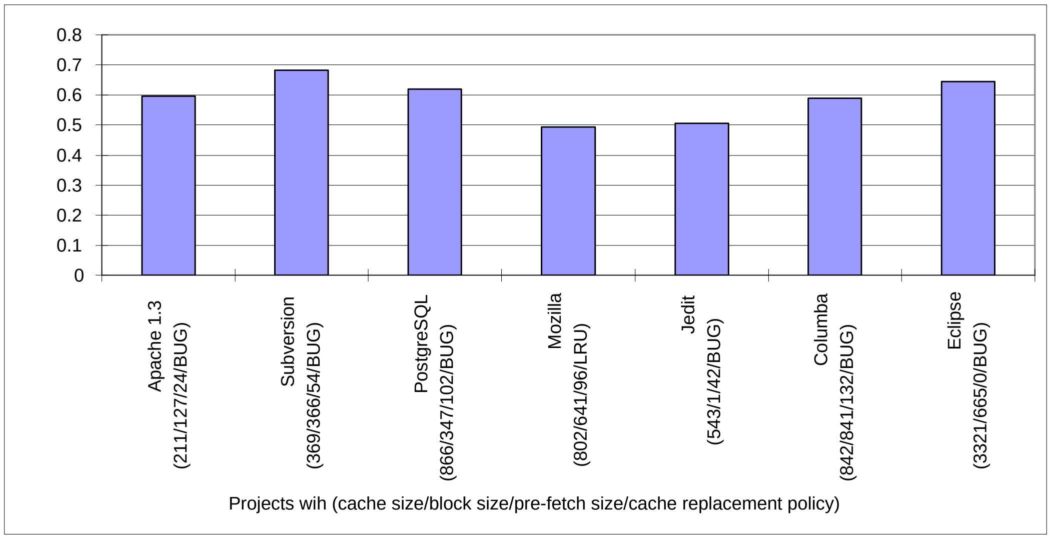

Figure 4-7. Hit rate of the entity level cache, with cache size set to 10% of all project entities.

Using a big block or pre-fetch size does not mean that each cache operation replaces all cached entities. For example, the Subversion cache size is 369 and block size (co-change entities) is 366. But when the co-changed entities are loaded into the cache, the bug cache tries to fetch entities one by one and a cache replacement algorithm is used to decide which cached entities should be replaced based on the properties of entities in the cache and an entity about to be added. For example, even if there are 366 new entities to be added into the cache (the cache size is 369) in one cache operation, the bug cache tries to add the 366 entities one by one by applying a cache replacement policy. Suppose the BUG cache replacement algorithm is used, and all entities in the cache have more than 1 bug. If there are 366 new entities to be added, and their bug count is all 0, then there is no cache replacement after the cache operation, since entities with 0 bug count cannot replace entities whose bug count is 1 or greater.

As expected, predicting bugs at the smaller grained entity level is more difficult than predicting bugs at coarser granularity. Besides the smaller target size, one possible reason why the entity level cache has a lower hit rate is that functions and methods change their names more frequently than files. Since the function/method name and its signature are used as an entity identifier, name changes lead to a loss of all previous history. This problem is discussed further in Section 4.4.4.

The bug cache approach is an on-line learning style of a prediction model, but there is still need to adjust cache options to yield optimal performance. The cache options yielding the best hit rates vary from project to project. For example, Apache 1.3 uses a big block size (127), while JEdit uses a small block size (1). This is due to the different distributions of bugs and changes across projects. An important implication is that bug prediction algorithms work best when they are adapted to a specific project [47].

4.3.2 Cache Replacement Policy

The bug cache approach uses three cache replacement algorithms, LRU, BUG and CHANGE. To see which algorithm works best for a given set of cache parameters, an analysis was performed using the same values for the cache size, block size, and pre-fetch size, varying only the cache replacement algorithm on each run.

Figure 4-4 and Figure 4-5 show the hit rates achieved using different cache replacement algorithms. At the file level, the LRU policy is good, working the best for 4 out of the 7 projects, with BUG being the best for the remaining 3. At the function/method level LRU works best only for the Mozilla project, with BUG being the best for the rest. Interestingly, the CHANGE policy works poorly at both granularities. This is somewhat contrary to the results in [23], where the most-frequently-modified heuristic was one of the best bug predictors.

4.3.3 Bug Localities

The four bug localities are the main factors used to load entities into the cache; however, are the contributions of the various localities the same? To measure the relative predictive strength of each kind of locality, an analysis was performed where each entity was marked with the reason for which it was loaded. After marking each entity with the reason for which it was loaded, the marks of the hit entities were observed. Figure 4-8 shows the ratio of loading reasons for hit entities of the Apache 1.3 project. The results show that bugs have strong temporal (59%) and spatial (18%) locality, and weak changed entity (4%) and new entity (1%) locality. The initial pre-fetch is surprisingly effective, accounting for 18% of the total hits.

One possible explanation for these results is that the existence of a bug is really the best predictor for other bugs in the same entity. That is, in most cases the effects of programmer's error are local to a single entity. However, there are enough cases where a programmer's error affects multiple entities, and hence spatial locality is useful for predicting them. When no bug data is available, code complexity, as represented by LOC, acts as a strong predictor of bugs. It is unclear whether changed and new entity locality is better than just randomly selecting entities.

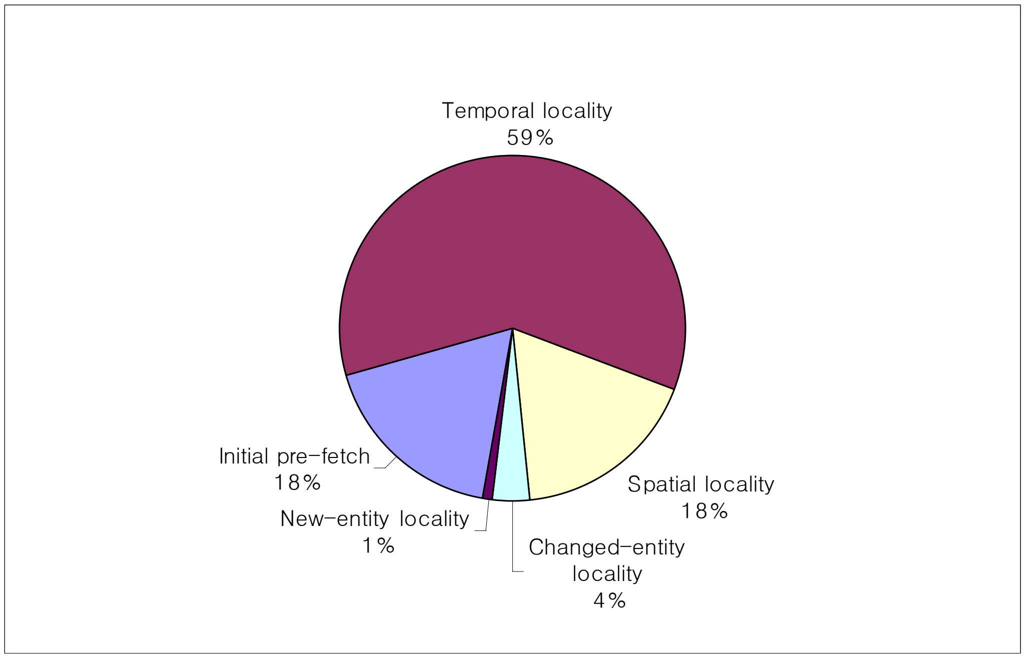

Figure 4-8. Entity level predictive contributions of initial pre-fetch and bug localities for Apache 1.3. The cache size is 211, block size is 127, pre-fetch size is 24, and BUG was used. The hit rate is 59.6%.

4.3.4 Bug Extraction and Hit Rate

Bug-fixes are identified by mining the text in SCM change logs [13, 58] using keywords as shown in Table 3-2. Most open source projects do not provide strong links between their bug tracking systems and their source code change histories. The number of identifiable bugs is decided by the quality of the change logs. For example, in the Subversion project, it is possible to recover 46% of existing fixes by mining the project change logs, and searching for the fix numbers used by their bug tracking system in the SCM commit comments.

Due to the limitations of mining software change logs, only partial lists of fixes and bug-introducing changes were retrieved. Suppose there is a project (like many commercial ones) that has perfect links between source code changes and their bug tracking system. This would make it possible to retrieve 100% of the fixes and bug-introducing changes from the project. Would this lead to an improved hit rate?

To see how the number of bug-introducing changes affects the hit rate, different numbers of fixes were obtained from the same project by using two different levels of data mining techniques (using different keywords). Using this approach, different numbers of fixes and bug-introducing changes were retrieved from the Apache 1.3 project, as shown in Table 4-3.

Table 4-3. Numbers of fixes and bugs using different levels of change log mining techniques from the Apache 1.3 Project.

Level	Keywords	Fixes	Bugs
Level 1	Bugfix, bug	215	463
Level 2	fix, bug, error	1,094	1,954

Using more strict keywords (level 1) only 251 fixes and 463 bugs were identified. Using less strict keywords (level 2) obtained 1,094 fixes and 1,954 bugs from the same project. The bug cache performances with various cache sizes using the two bug datasets were analyzed. Resulting hit rates are shown in Figure 4-9. The level 2 dataset has a better hit rate than level 1, by about 3-7%. This result suggests that as the quality of the bug dataset improves and there is a greater number of bugs to use with the algorithm, the hit rate improves. In other words, if more bugs are identified, the identified bugs have better localities.

A good dataset of bug-introducing changes is no guarantee of high hit rates. The Columba dataset is of very high quality due to a diligent development team. However, its hit rate at the file level is a respectable, but not stunning 83%. Eclipse has a greater hit rate (95%) with a lower quality dataset.

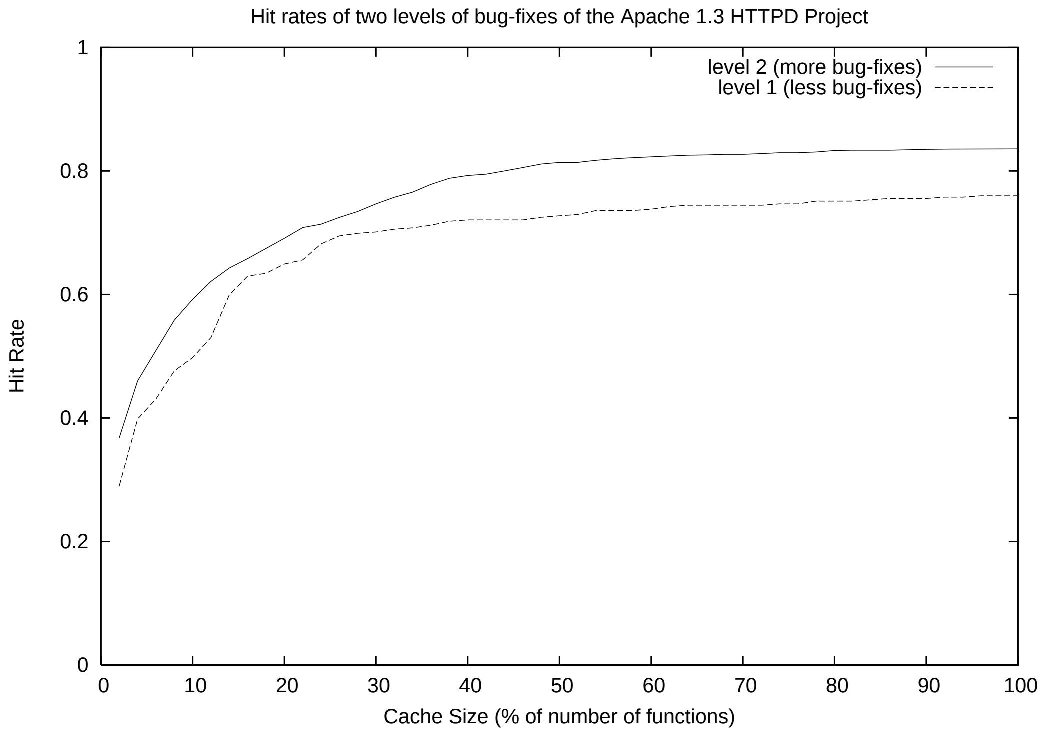

Figure 4-9. The level 1 and level 2 hit rates of the Apache 1.3 project with various cache sizes. The block size is 10% of the cache size, and no *pre-fetches*. LRU is used. Note that the hit rate using 100% cache size is not 100%, since there is no pre-fetch and there are compulsory misses – first seen bugs.

4.3.5 Cache Size vs Hit Rate

It is important to keep cache size small to improve the utility of the cache list. If the bug cache is large, the hit rate will improve since a larger fraction of the entire code base is in the cache. As the cache gets larger, its precision drops which reduces its utility. What is a reasonable tradeoff between cache size and hit rate?

Figure 4-9 is a typical graph of cache size vs. hit rate, showing strong initial growth that quickly tapers off in a long-tailed asymptote. The cache-based bug prediction work in [23] also shows a similar curve. In general, one wants to capture as much of the initial period of strong growth as possible, while minimizing the size of the cache. There are two approaches. An organization can pick a cache size that reflects the available resources for intensive quality assurance, and accept the resulting hit rate. Alternately, an organization can pick a desired hit rate, and choose a cache size that achieves it. For example, if the Apache Software Foundation wanted to maximize their hit rate, they should pick a cache size of 30-40%, since in Figure 4-9 it shows that this encompasses the majority of hit rate growth.

4.3.6 Cached Entities (10%) and LOC

In most of this analysis, a cache size of 10% of project entities is used. For example, Apache 1.3 has 2,113 functions so the cache size has been set at 211 (10% of the total function numbers). However, 10% of functions do not necessarily mean the cache holds just 10% of the total lines of code. In fact, the higher LOC count is expected, since longer entities would be expected to have more bugs just due to size alone. To get a sense of the code contained in the cached entities, cached entity LOC was observed after the final cache operation and compared with the total entity LOC. The cached entity LOC varied based on the cache options. Figure 4-10 shows cached entity LOC with the best bug cache options for each project.

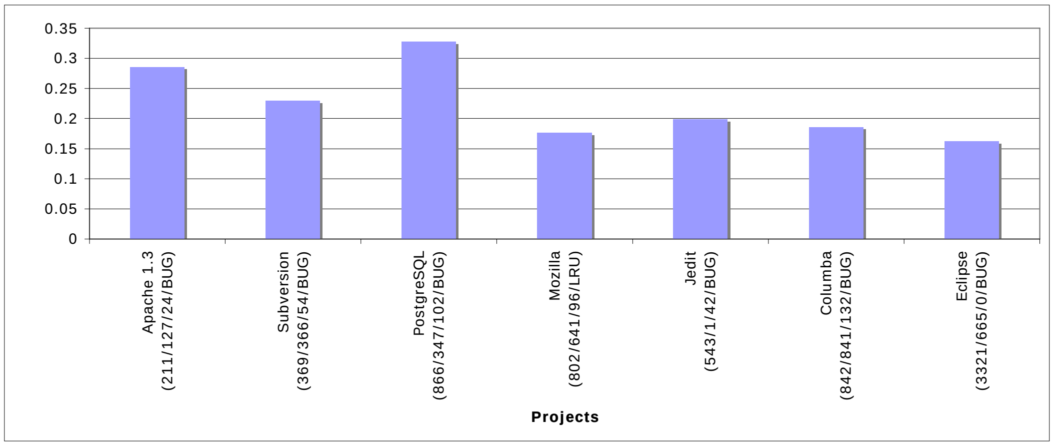

Figure 4-10. Cached function LOC/total LOC

Even though only 10% of the total entity numbers are cached, cache entity LOC is about 16-33% of the total LOC. Prior research [23, 50] also uses percentage of entities such as 20% of files or top 10, but did not measure LOC or describe the LOC of entities in the list.

4.4 Discussion

4.4.1 Applications

Right now, the bug cache operates in a lab environment. However, its implementation can be put to use in various ways:

- A typical application of the bug cache is as follows: Whenever a bug is found and fixed, a tool can automatically identify the last change to the original code and then update the bug cache from the moment this bug-introducing change was applied—that is, from the moment the bug was introduced. A manager or developer can then use the list for quality assurance—for example, they can test or review the entities in the bug cache with increased priority.
- While bugs can only become part of the bug cache at the time they are fixed, the bug cache still contains suspicious locations based on recent changes. In particular, the bug cache would direct resources to newly added or changed locations. These locations also have a higher chance of containing bugs.
- Developers can also directly benefit from the bug cache. If a developer is working on entities in the cache list, he or she can be made aware that he or she is working on a potentially instable or buggy part of the software.

All in all, a bug cache will help direct efforts to those entities that are most likely to contain bugs and thus increase quality, and reduce effort.

4.4.2 Bug Cache Model

Why does the bug cache model have better predictive power than previous prediction models? Most models found in the literature use bug correlated factors and developed a model to predict future bugs. Once developed, the model is static, and incorporates all previous software change history data and factors. The bug cache model is dynamic, and is able to adapt more quickly to recent software change history data, since the bug occurrences directly affect the model. For example, when a bug occurs and the cache model misses the bug, it loads the entity regardless of the cache parameters or cache replacement policy.

The cache model continually receives feedback (hits and misses) from its predictions and adapts its model based on the prediction results. One can view the cache as receiving a penalty (fetch entities) for a miss, or a reward (keep entities) for a hit. This approach is similar to on-line machine learning algorithms [2]. The bug cache learns from the bug distributions of each project. Even though each project may have different bug distributions, it can adaptively learn from hits and misses and update its prediction model for its next prediction. This adaptive approach yields better predictive power.

The selection of cache options and cache replacement policies affects the cache hit rate. A brute force analysis is used to discover the best cache options for each project. Note that the cache options for the best hit rates vary due to the different bug and change distributions across projects. Despite this variation, is it possible to develop a rule of thumb for a good cache size, block size, and pre-fetch size parameters? Based on the observations, about 7-15% of the total number of files would be a good cache size. For the function/method level, a block size of 30-50% and a pre-fetch size of 10-30% of the cache size tend to work well. Projects using a bug cache need to periodically optimize their cache options by performing a brute force analysis of the various parameters to see which combination yields the best hit rate.

4.4.3 Fixes vs Bug-introducing Changes

In this study, bug-introducing changes are used—in contrast to most prior studies, which use fixes. In fact, fixes are very different from bug-introducing changes since a fix is a change that fixes a bug.

In the bug cache experiments, revision data gathered from revisions *1* to *n-1* are used to predict bugs at revision *n* but, sometimes bug-introducing changes at revision *n-1* are identified by a fix that occurred at revision *n+x* [58]. In practice, finding all bug-introducing changes from revision *1* to *n-1* at revision *n* is not feasible. Therefore, the set of bug-introducing changes known at revision *n* and time *t* is smaller than the set of bug-introducing changes at revision *n* known at a later time *t+x*. This growth in knowledge is due to the fact that at any given revision there may be bugs that were introduced, but not yet discovered and fixed.

To see how a bug-fix based cache would work, a bug cache analysis was performed over the same set of seven projects, applying a brute force technique to select the best cache parameters for each project (with cache size 10%, file level) using bug-fix (not bug-introducing) data. Results are shown in Figure 4-6. Overall, the general trend is a 2-4% drop in accuracy as compared to the hit rates using bug-introducing changes shown in Figure 4-11. The Eclipse and Subversion hit rates remain almost the same.

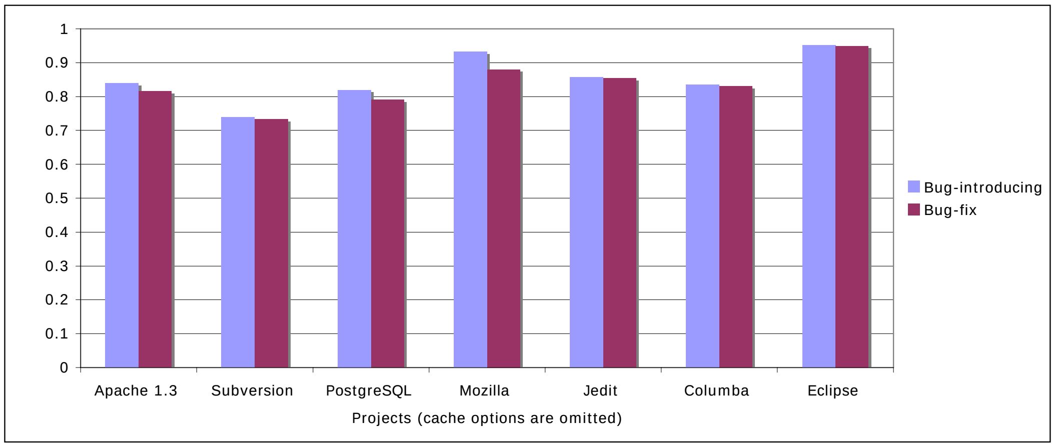

Figure 4-11. Hit rate of the file level cache using bug-introducing and bug-fix data, with cache size set to 10% of all project files.

These results indicate that projects can still achieve strong hit rates even when using only bug-fix data (fix-based cache). The reason for this is that fixes also have temporal locality, since bug-introducing changes and their fix times are correlated. For example, a bug can be fixed after the bug was introduced.

Fixes have strong spatial locality, since fix changes are grouped by a revision and all fixes in a revision are also co-changes (all changes in a revision are fixes, if the revision is a fix revision). Zimmermann et al. shows that there are strong association rules in co-changes [70]. A fix-based cache can be used in practice since, using the cached entities, it is possible to predict fixes before they happen—predicted fix changes include bugs before they are fixed. While the bug-introducing based cache provides theoretical evidence of localities of bug occurrences, the fix-based cache can be used for practical use.

4.4.4 Origin Analysis

For programming languages that allow function/method over-riding, it is possible that more than one function/method has the same name in a project. To uniquely identify entities, an identifier comprised of its file name, function name, and signature is used. One consequence is that when entities change their names or signatures, all of their previous history is lost.

For example, assume the '*foo*' function evolves as shown in Table 4-4. The argument type has been changed in version 2, causing its entity identifier to change as well. Suppose the function has bugs in versions 1 and 3, and is not initially in the cache. There will be one miss for the bug in version 1, and another miss for the bug in version 3 since the function's identifier was changed at version

2, causing it to be dropped. If it is possible to follow signature changes or function/method name changes, there would only be one miss for the bug in version 1, and the bug in version 3 would be a hit. Ongoing research on origin analysis [19, 30] offers hope in this area, since it provides the ability to track a single function or method across multiple names and signature changes. Applying origin analysis seems very likely to increase the hit rate of the bug cache, especially at the entity level.

Table 4-4. An example evolution of function foo.

Version	1	2	3
Function	foo(int x)	foo(String x)	foo(String x)
Bug	Yes	No	Yes

4.5 Summary

If we know that a bug has occurred, it is useful to search its vicinity for further bugs. The bug cache model predicts these future bugs with high accuracy: at the file level, it can cover about 73-95% of future bugs; at the function/method level, it covers 49-68% of future bugs—with a cache size of only 10%. This is a significantly better accuracy and lower granularity than found in the previous state of the art. The cache can serve as a priority list to test and inspect software whenever resources are limited (i.e. always).

The bug cache is able to adapt more quickly to recent software change history data, since the bug occurrences directly affect the model. This is another significant advantage over static models. It is the first approach to use spatial locality as a bug predictor, and the combination of the four locality concepts has significant advantages.

4.6 Future Work

There is still room for improvement. The future work for bug caching will concentrate on the following topics.

- The bug cache approach explored four bug localities. There might be more localities, such as special locality based on data dependence or slicing information based spatial locality. Identifying more bug localities and exploring them will be future work.
- In this study, option combinations for each project varied due to the various bugs or change distributions of each project. Developing self-adaptive algorithms for bug cache option finding is another avenue of future study. Such algorithms will learn from cache hits and misses and adjust cache options to optimize performance.
- Different levels of software granularity result in different hit rates. The *hierarchical caches* that fetch entities at different granularities such as modules, files, and methods may be useful. Depending on its usage, the cache can provide different levels of buggy entity lists.

- Finally, integrating the bug cache into history-aware programming tools such as eROSE [70] is useful. This way, whenever a bug is fixed, the tool can automatically suggest future locations to be examined for related bugs.

5 Buggy Change Classification

Assume you are a software developer working on an ongoing software product. Whenever you submit a source code change, you may wonder whether your change has introduced a new bug. If a tool could tell you that your change is buggy, you could inspect your change carefully; otherwise, you would assume that your change is error-free. This tool acts to classify your change into one of two states: *buggy* or *clean*. If such an automatic software change classification tool can predict changes with acceptable accuracy, the tool can be used in the software development process as follows: make a change, receive feedback from the tool, inspect the change, and then make another change.

This chapter explores the hypothesis that it is possible to develop a prediction model with acceptable accuracy using machine learning algorithms and information from existing software histories. To test this hypothesis, the file change histories were extracted from the software configuration management repositories of 12 open source projects, the changes were classified using two machine learning classification algorithms, and the classifications were evaluated for accuracy.

Looking at the history of an individual project file one sees a series of revisions separated by changes. Each change either (a) modifies the code or (b) modifies the code and introduces a bug. The term, *buggy change* (b) is used to mean a change that caused one or more bug-fixes (fixes). In contrast, if a change does not cause any fix, it is a clean change (a). Buggy and clean changes are defined as Boolean values that are opposite end-points of the same scale.

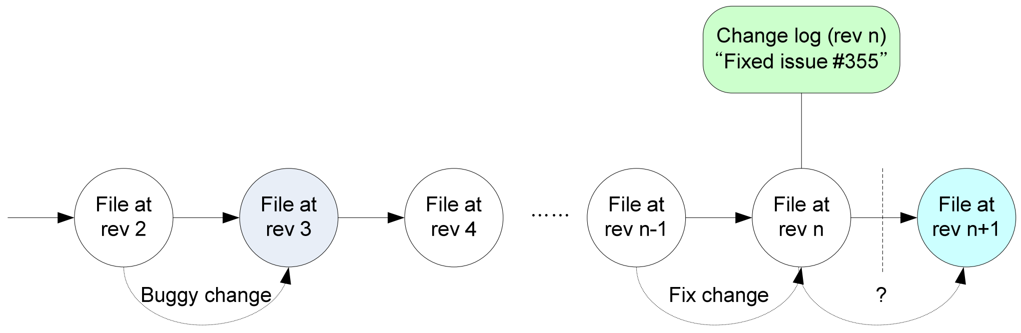

Figure 5-1. Example buggy and clean file changes.

Following the process outlined in Chapter 3, every change is labeled as *buggy* or *clean* in 12 open source projects. Using these labeled changes (instances) and two machine learning algorithms, Naïve Bayes [54] and Support Vector Machine [26], a change classifier is trained. Once trained, the classifier is used to "predict" future buggy changes. For example, in Figure 5-1, after training a change classification model using change data from revisions *1* to *n*, if there was a new and unknown change (i.e. revision *n+1*), it is possible to classify this change as either buggy or clean using the trained classification model. This act of classification has the effect of predicting which changes have bugs.

An automatic software change classifier has multiple uses. After a new change is committed, the classifier can send an alert to the developer if the change has been classified as buggy. By integrating the classifier into a software development environment (SDE), the SDE can monitor source code changes, and as soon as a change is identified as buggy – perhaps even before an SCM commit – developers can be notified. The buggy change notification can raise the awareness of developers on whether they are creating bugs, so that they can inspect or review the change carefully, or possibly adopt a more conservative coding practice. This awareness may lead to bug-fixes before a bug has even been reported.

Machine learning (supervised learning) algorithms learn classification models from a training dataset. A training dataset is a set of instances that consists of multiple features that have been labeled with their correct class (buggy or clean). The trained classification model can classify unknown instances (testing dataset) with features but no labels. The training dataset is the set of changes extracted from SCM repositories, with each change containing multiple features, and labeled as either clean or buggy.

In the remainder of this chapter, Chapter 5.1 overviews the change classification approach. The feature generation and classification algorithms (Chapter 5.2) are explained, and then the change classification experimental setup is described, including a description of the corpus and methods used for measuring accuracy (Chapter 5.4). Following are results from change classification at the file change level (Chapter 5.6), along with informative feature identification. Chapter 5.7 provides discussion of these results. Chapter 5.8 summarizes the change classification approach.

5.1 Overview of Change Classification Approach

This chapter provides an overview of the process of creating a corpus, a set of labeled instances from the 12 open source projects listed in Table 5-1, how to train machine learning classifiers, classification of changes, and evaluation of each classifier.

Table 5-1. Analyzed open source projects. The period shows the change history time spans of the analyzed projects. The *# of revisions* column indicates the number of revisions extracted. For the Eclipse project, only the jdt.core module is examined due to the large size of the entire project. Similarly, the mozilla/browser/components module is explored for the Mozilla project.

Project	Period	# of revisions
Apache HTTP 1.3 (A1)	10/1996 ~ 10/2005	7,777
Bugzilla (BUG)	08/1998 ~ 10/2005	4,462
Columba (COL)	04/2001 ~ 10/2005	2,841
Gaim (GAI)	03/2000 ~ 10/2005	8,688
GForge (GFO)	08/2001 ~ 10/2005	1,681
Jedit (JED)	09/2001 ~ 11/2005	1,449
Mozilla (MOZ)	04/2002 ~ 10/2005	1,606
Eclipse JDT (ECL)	06/2001 ~ 11/2005	11,117
Plone (PLO)	02/2002 ~ 10/2005	2,226
PostgreSQL (POS)	07/1996 ~ 09/2005	7,622
Scarab (SCA)	12/2000 ~ 10/2005	8,026
Subversion (SVN)	08/2001 ~ 10/2005	7,297

The Support Vector Machine (SVM) and Naïve Bayes classifiers implemented in the Weka Toolkit [65] are used in this study. These two classifiers are employed since they are widely used for text classification and have different classification characteristics. The discriminative power of each feature is compared to find informative features for performing change classification. The basic process is as follows:

Creating a Corpus:

1. Extract file changes from 12 open source project histories using Kenyon [6].
2. Identify the fix changes for each file by mining the project's revision history, as stored in its SCM repository. The techniques described in [13, 58] are used for this step.
3. Identify bug-introducing (or fix-inducing, a term used in [58]) changes at the file level by analyzing the revision history and SCM annotation information, as described in [58]. All bug-introducing changes are labeled as *buggy* and the other changes are labeled as *clean*.
4. Extract features from the file level change information, including the complete source code, the lines modified in this change (delta), and change metadata such as author and change time.

By the end of step 4, a corpus, a set of labeled instances has been created of each file change. Each instance is a set of features [2].

Classification:

5. Using the corpus, classifiers are trained for each project.

6. To evaluate classification performance, the 10-fold cross-validation method [45] is used. Standard classification evaluation measures of accuracy, recall, precision, and F-value are computed.

Informative Feature Identification:

7. Subsets of the corpus are created by combining features and performing steps 5-6 to compare classification performance using different feature combinations.

8. Using the chi-square test statistic [9], to compute degree of correlation to classes, the ranks of each feature care are computed to determine the most informative features for change classification.

Overall, the Support Vector Machine (the best classifier in this study) classified file changes as either buggy or clean with 78% accuracy on average (ranging by project from 65%-93%) and 65% buggy change recall on average (43%-98%).

The prediction accuracy reported in this chapter is comparable with existing research. In the literature, file level bug prediction accuracy using object-oriented metrics and machine learning algorithms is approximately 70% [22]. Ostrand et al.

can identify the top 20% of the problematic file list, and by using the list they can predict bugs with 71-93% accuracy [50]. My research differs from this previous bug prediction work in that I:

Classify changes: Most previous work focuses on finding prediction or regression models to identify bug-prone or buggy modules, files, and functions [20, 46, 49]. The change classification approach has multiple benefits, including prompt and fine-grained prediction. It is possible to predict buggy changes at the file level as soon as a change is made. It is easier to inspect recent changes than those made a long time ago. Prediction is very fine-grained. The prediction granularity is individual file changes, comprised of an average of 20 lines of code, as compared to the related work that localizes to an entire file, which in the case of the corpus, had an average length of 300 lines of code (see Table 5-8).

Use bug-introducing changes: Most previous research uses bug-fix data when making predictions or validating their prediction models. One advantage to the change classification approach is that it uses bug-introducing changes, and hence it is possible to know when the bugs occurred. Bug-fix data only indicates where the bug occurred, not when it was introduced or who introduced it. Only bug-introducing changes make it possible to label changes as buggy or clean.

Use features from source code: Existing research that mines software history data to perform bug prediction uses complex metrics [27], accumulates change/bug counts [20, 43], or code churn [46]. When selecting predictors, researchers usually do not take advantage of the information provided by the source code text itself, and thereby miss a valuable feature. In contrast, every term in the source code—every variable, method call, operator, constant, comment text and more—are used as features to train change classification models.

Are independent of programming languages: The change classification approach is programming language independent, since the bag-of-words method [56] is used to extract features from source code changes. The analyzed projects span many popular current programming languages, including C/C++, Java, Perl, Python, Java Script, PHP, and XML. The change classification approach obtains comparable accuracy, buggy change precision, and recall from all the projects despite their different programming languages, and even though no semantic understanding of the source code is used.

5.2 Algorithms

To classify software changes using machine learning algorithms, the first step is to train a classification model using buggy and clean change features. Change information available from a project’s SCM repository includes multiple feature sources such as each change’s log message, source code, change delta, author, and commit date. Features, such as author and commit date can be generated directly from the change information, but the other features, such as terms in the source code, require further feature generation techniques. A modified bag-of-words approach is used to generate features from source code files and change log texts. To extract the C/C++/Java source code complexity metrics, the Understand C/C++ and Java tools [55] are used (no complexity metrics were generated for PHP, Perl, Python and XML source code due to the lack of available tools for computing the complexity measures for these languages).

This chapter discusses feature generation techniques and introduces the classification algorithms.

5.2.1 Feature Generation

A well known characteristic of machine learning classification algorithms is that their accuracy is sensitive to the selection of features [21, 56]. Fortunately, humans are not required to provide a list of known good features, since a feature selection algorithm can automatically find good features out of a mixed set of relevant and irrelevant features [65]. A classifier's accuracy will not suffer from large numbers of features, or redundant features. All possible features are extracted from file change information, and fed into a feature identification process to determine the important features for change classification [41, 65].

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 deleted code (deleted delta) between the two revisions. A file change has associated metadata, including the change log, author, and commit date. By mining change histories, it is possible to derive features such as co-change counts to indicate how many files are changed together in a commit, the number of authors, and the previous change count of a file. Every term in the source code, change delta, and change log text is used as a feature. The detailed feature generation methods are described below.

5.2.2 Feature Generation from Change Metadata

File change metadata is a good feature source for predicting buggy changes. There are 8 features from change metadata: 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. In other research, cumulative bug and change counts are commonly used as bug predictors [20, 43, 49, 51, 58, 60, 70].

5.2.3 Complexity Metrics as Features

Software complexity metrics are also commonly used to measure software quality and identify risky modules [22, 27]. A range of traditional complexity metrics of source code is computed using the Understand C/C++ and Java tools [55]. As a result, 61 complexity metrics (every complexity metric these tools compute) were extracted for each file including LOC, lines of comments, cyclomatic, and max nesting. Since there are two source code files involved in each change (old and new revision files), the delta of each complexity metric value is available. The complexity metric deltas are an important feature since they indicate whether or not a file change increases or decreases in complexity. The 61 complexity metric deltas are used as features as well.

5.2.4 Feature Generation from Change log, Source Code, and File names

Change logs are similar to email or news articles in that they are human readable texts. Each word in the change log carries a semantic meaning. Feature engineering from texts is a well studied area, with bag-of-words, latent semantic analysis (LSA), and vector modeling being widely used approaches [53, 56]. Among them, the bag-of-words (BOW) approach, which converts a stream of characters (the text) into a bag of words (index terms), is simple and performs fairly well in practice [56, 57]. BOW is used to generate features from change logs.

All words were extracted, except for special characters, and were converted to lowercase. The existence (binary) of a word in a document is used as a feature. Every term in the source code is used as a feature, including operators, numbers, keywords, and comments. To generate features from source code, a modified version of BOW, called BOW+, is used to extract all normal terms. BOW+ additionally extracts operators, since operators such as "*!=*", "*++*", and

“&&“ are important terms in source code. The BOW+ extraction is used on added delta, deleted delta, and new revision source code.

The directory and file name are converted into features, since they encode both module information and some behavioral semantics of the source code. For example, the file (from the Columba project), *‘RecieveOptionPanel.java’* in the directory, *‘src/mail/core/org/columba/mail/gui/config/account/’* reveals that the file received some options using a panel interface, and the directory name shows the source code is related to *‘account’*, *‘configure’*, and *‘graphical user interface’*. Some researchers perform bug predictions at the module granularity by assuming that bug occurrences in files in the same module are correlated [20, 23, 60]. The BOW approach is used by removing all special characters, such as slashes, and extracting the words from the directories and file name. Directories and file names often use Camelcase [63]. For example, ‘*RecieveOptionPanel.java*’ combines ‘*receive*’, ‘*option*’, and ‘*panel*’. To extract such words correctly, a case change in a directory or file name is considered as a word separator. This method is defined as BOW++. Table 5-2 summarizes features generated and used in this study.

Table 5-2. Feature groups. Feature group description, extraction method, and example features.

Feature Group	Description	Extraction method	Example Features
Added Delta (A)	Terms in the added delta source code	BOW+	if, while, for, ==
Deleted Delta (D)	Terms in the deleted delta source code	BOW+	true, 0, <, ++, int
Directory/file Name (F)	Terms in the directory/file names	BOW++	Src, module, java
Change Log (L)	Terms in the change log	BOW	fix, added, new, feature
New Revision Source Code (N)	Terms in the new revision source code file	BOW+	if, ||, !=, do, while, string, false
Metadata (M)	Change metadata such as time and author	Direct	author: hunkim, commit hour: 12
Complexity Metrics (C)	Software complexity metrics of each source code	UD tools	LOC: 34, Cyclomatic: 10

5.3 Classification Algorithms

There are many text classification algorithms. Naïve Bayes [54] and Support Vector Machines [26] are used in this experiment for two reasons: 1) they are widely used in text classification applications; and 2) they are very different from each other. Naïve Bayes models how documents in each class are generated and derives the decision boundary from the generative models of different classes, while Support Vector Machine is a discriminative model that directly models the decision boundary between classes.

5.3.1 Naïve Bayes

Naïve Bayes is a multi-class classification algorithm, where the class labels can take several different values [54]. In this study case, two classes, buggy and clean, are used. For each change that is represented as a sequence of features, assume its length is independent of its class, and the probability of generating each feature is independent of other features in the change. If the change belongs to class c_j, a multinomial distribution $P(w_v|c_j)$ is used to model the generation process of the feature sequence, where $P(w_v|c_j)$ is the probability of generating a feature w_v given class c_j. Given a set of training changes with class labels, the maximum estimation of $P(w_v|c_j)$ is the total number of occurrences of features w_v in class j divided by the total number of feature occurrences in class j.

Let $P(c_j)$ be the probability of a random change belonging to class j. The maximum likelihood estimation of $P(c_j)$ is the number of changes in class j divided by the total number of changes.

The probability of a change $d_i=(w_1, w_2, ..., w_N)$ given its class label c_j is:

$P(d_i|c_j)= C(d_i)\ P(w_1|c_j)\ P(w_2|c_j)\ ...\ P(w_N|c_j)$

Where $C(d_i)$ is a value that only depends on d_i. Using the Bayesian Rule, it is possible to estimate the probability that a testing change d_i belongs to class c_j:

$$P(c_j \mid d_i) = \frac{P(d_i \mid c_j)P(c_j)}{P(d_i)} \propto P(c_j)P(w_1 \mid c_j)P(w_2 \mid c_j)...P(w_N \mid c_j)$$

The most likely class for the change d_i is the class c_j that gives the highest value of $P(c_j|d_i)$.

5.3.2 Support Vector Machines

Support Vector Machines were originally designed for binary classification, where the class label can take only two different values. A support vector machine tries to find the maximum margin hyperplane, a linear decision boundary with maximum margin between the boundary and the training examples in class 1 and training examples in class 2 [62]. This hyperplane gives the greatest separation between the two classes. The task of finding this hyperplane can be converted into a constraint minimization task:

$$\max_{\beta,\beta_0,|\beta|=1} C\text{, } \textit{subject to } y_i(x_i^T\beta+\beta_0) \ge C,\ i=1,...,N$$

Where x_i is a vector that represents the training example and each dimension of the vector corresponds to the weight of an indexing feature. y is a scalar usually used to represent the value: y_i=1 if x_i belongs to class 1, and y_i=-1 if xi belongs to class 2. β is a vector, β_0 is a scalar, and (β,β_0) defines the hyperplane. C is called the margin: the distance from the hyperplane to either class. The solution can be found efficiently, and the Sequential Minimal Optimization (SMO) algorithm described in [52] is used. The SMO implementation in the Weka Toolkit was used in this study [65]. There are several variations of SVM, and much literature about SVM exists. A detailed discussion of SVM is beyond the scope of this dissertation, and [26] has more details.

The performance of the two algorithms is discussed in Chapter 5.7.

5.4 Experiment Setup

5.4.1 Corpus Information

Since different projects have different durations, file change features from revisions 500-1000 (or revisions 500-750 for big projects) are used in this study. Revisions 500-1000 are selected, since the change patterns in the first part of a project (revisions 1-500) may not be stable. Due to the inherent flux in the start of a project, the changes in this period may have many refactorings and abnormal changes. In the literature, some researchers ignore the first part of a project's history to remove potential abnormalities in change patterns [23]. The classification results of the first period (revision 1-500) and the second period (revision 500-1000) are discussed in Section 5.7.3.

Using the feature engineering technique described previously, features were generated from all file changes in the analyzed range of revisions. Each file change is represented as an instance, a set of features. Using the bug-introducing change identification algorithm, each instance was labeled as either clean or buggy. Table 5-3 summarizes the corpus information.

For example, for Apache 1.3 (A1), 500 revisions which were made between 10/1996 and 01/1997 were analyzed. During this period, there were 700 total file changes, of which 566 were clean and 134 were buggy (19.1%). The count of all unique features extracted from these 700 changes is 11,445.

Table 5-3. Summary of the corpus information. * is the average of percentages of buggy changes.

Project	Revisions.	Period	# of clean changes	# of buggy changes	% of buggy changes	# of features
A1	500-1000	10/1996-01/1997	566	134	19.1	11,445
BUG	500-1000	03/2000-08/2001	149	417	73.7	10,148
COL	500-1000	05/2003-09/2003	1,270	530	29.4	17,411
GAI	500-1000	08/2000-03/2001	742	451	37.8	9,281
GFO	500-1000	01/2003-03/2004	339	334	49.6	8,996
JED	500-750	08/2002-03/2003	626	377	37.5	13,879
MOZ	500-1000	08/2003-08/2004	395	169	29.9	13,648
ECL	500-750	10/2001-11/2001	592	67	10.1	16,192
PLO	500-1000	07/2002-02/2003	457	112	19.6	6,127
POS	500-1000	11/1996-02/1997	853	273	24.2	23,247
SCA	500-1000	06/2001-08/2001	358	366	50.5	5,710
SVN	500-1000	01/2002-03/2002	1,925	288	13.0	14,856
Total	N/A	N/A	8,272	3,518	29.8*	150,940

Note that the buggy change percentage of Bugzilla (BUZ) is relatively higher than that of other projects. Since Bugzilla is a bug tracking project, the project heavily uses Bugzilla to keep the project issues. Since the fix identification algorithm used in this dissertation relies on the linkage between log messages and bug tracking numbers as shown in Table 3-1, the algorithm identifies many fix changes in the Bugzila project. Many identified fixes cause identification of many buggy changes. Similarly, Scarab (SCA), another issue tracking system, has higher buggy change percentage.

After they are extracted from changes, features are grouped based on the feature sources and analysis is performed to identify important feature groups. The feature groups and the number of features in each group are summarized in Table 5-4.

Table 5-4. Feature groups. The numbers of each feature group are shown.

Feature Group	Number of features of projects											
	A1	BUG	COL	GAI	GFO	JED	MOZ	ECL	PLO	POS	SCA	SVN
Added Delta (A)	2024	2506	3811	2094	1895	2939	3079	2558	1540	3532	1290	2663
Deleted Delta (D)	1610	1839	3227	1956	1832	2352	2176	2200	1073	2995	836	2117
Directory/File Name (F)	93	66	559	39	242	377	105	456	221	472	106	195
Change Log (L)	1257	1124	869	1094	3970	431	959	53	2835	1161	650	2474
New Revision Source Code (N)	6330	4604	8814	3967	4481	7649	7320	10794	2671	14956	2697	7276
Metadata (M)	8	8	8	8	8	8	8	8	8	8	8	8
Complexity Metrics (C)	122	0	122	122	0	122	0	122	0	122	0	122
*Total	11445	10148	17411	9281	8996	13879	13648	16192	6127	23247	5710	14856

5.4.2 Feature Selection

As described in Table 5-3, each project has a large number of features. For example, Columba has 17,411 features and PostgreSQL has 23,247 features. There are various ways to select important features and reduce the number of features without sacrificing or even improving classification accuracy [21, 41]. This study tries to identify important feature groups and individual features, so no feature selection algorithms are applied.

5.5 Evaluation Measure

There are four possible outcomes from using a classifier: classifying a buggy change as buggy ($n_{b\to b}$), classifying a buggy change as clean ($n_{b\to c}$), classifying a clean change as clean ($n_{c \to c}$), and classifying a clean change as buggy ($n_{c \to b}$).

The accuracy, recall, precision, and F value measures are widely used to evaluate classification results [57, 67]. These measures are used to evaluate the file change classifiers, as follows [2, 44, 67]:

$$\text{Accuracy} = \frac{n_{b \to b} + n_{c \to c}}{n_{b \to b} + n_{b \to c} + n_{c \to c} + n_{c \to b}}$$

That is, the number of correctly classified changes over the total number of changes.

$$\text{Precision (bug)}, P(b) = \frac{n_{b \to b}}{n_{b \to b} + n_{c \to b}}$$

This represents the number of correct classifications of the type ($n_{b \to b}$) over the total number of classifications that resulted in a bug outcome.

$$\text{Recall (bug)}, R(b) = \frac{n_{b \to b}}{n_{b \to b} + n_{b \to c}}$$

This represents the number of correct classifications of the type ($n_{b \to b}$) over the total number of changes that were actually bugs.

$$\text{F-value (bug)} = \frac{2 * P(b) * R(b)}{P(b) + R(b)}$$

This is a composite measure of precision and recall.

Similarly, we can compute clean change recall, precision, and F-value:

Precision (clean), $P(c) = \dfrac{n_{c\to c}}{n_{c\to c} + n_{b\to c}}$

Recall (clean), $R(c) = \dfrac{n_{c\to c}}{n_{c\to c} + n_{c\to b}}$

F-value (clean) = $\dfrac{2 * P(c) * R(c)}{P(c) + R(c)}$

5.5.1 Validation Method

Among the labeled instances in a corpus, some subset must be used as a training set or a test set, since this affects classification accuracy. The10-fold cross-validation technique [45, 65] is used to handle this problem. 10-fold cross-validation works as follows: first, we randomly divide a corpus into 10 folds of the same size, as shown in Figure 5-2. The first fold is selected as a test set, and the others as a training set. A classification algorithm uses the training and test set to train and test the model. The method iterates by selecting the 2nd, 3rd, … 10th fold as a test set, and the others as a training set. After 10 iterations, accuracy measures can be computed by summing up the classification results.

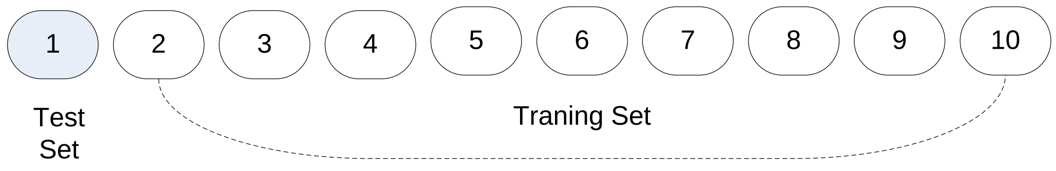

Figure 5-2. Operation of the 10-fold cross validation.

Training and classification is performed 10 times with each fold tested exactly once as a test set. All accuracy measures reported in this chapter are computed using 10-fold cross-validation.

5.6 Results

This chapter presents change classification results including accuracy, recall, precision, and F values. The important feature groups and individual features are identified.

5.6.1 Change Classification Accuracy

Figure 5-3 shows accuracy and buggy change recall of the 12 projects using all features listed in Table 5-3. Change classification accuracy ranges between 63% and 92%, varying by projects. Buggy change recall ranges between 43% and 86%. Buggy change precision ranges between 44% and 85%.

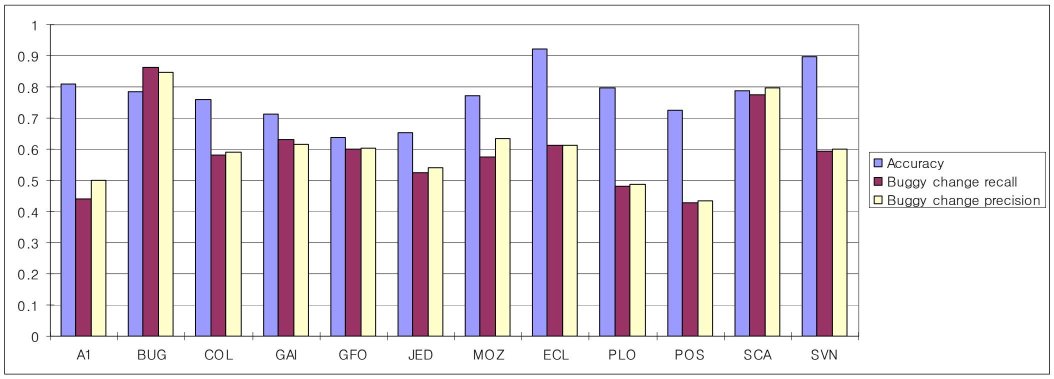

Figure 5-3. Change classification accuracy, buggy change recall, and buggy change precision of the 12 analyzed projects using SVM and all features.

Detailed recall, precision, and F values are reported in Table 5-5. The recall indicates among all real buggy changes how many can be caught by the change classification approach. The precision indicates among the predicted buggy changes how many of them are real buggy changes.

Since this study is the first to classify file changes, there are no directly equivalent projects which can be used to compare results. It is only possible to evaluate the change classification performance by comparing it with other research that has performed bug prediction at the file or module level, despite the fact that they are not directly comparable.

Table 5-5. Change classification accuracy, recall, precision, and F values of 12 open source projects. The SVM classification algorithm is used with all features.

Project	Accuracy	Bug recall	Bug precision	Bug F	Clean recall	Clean precision	Clean F
A1	0.809	0.44	0.5	0.468	0.896	0.871	0.883
BUG	0.784	0.863	0.847	0.855	0.564	0.596	0.579
COL	0.758	0.581	0.59	0.586	0.831	0.826	0.829
GAI	0.712	0.632	0.617	0.624	0.761	0.773	0.767
GFO	0.637	0.599	0.602	0.601	0.669	0.666	0.668
JED	0.653	0.525	0.540	0.532	0.730	0.719	0.724
MOZ	0.773	0.574	0.634	0.602	0.858	0.825	0.841
ECL	0.921	0.612	0.612	0.612	0.956	0.956	0.956
PLO	0.798	0.482	0.486	0.484	0.875	0.873	0.874
POS	0.726	0.429	0.435	0.432	0.822	0.818	0.820
SCA	0.786	0.776	0.796	0.786	0.796	0.777	0.786
SVN	0.896	0.594	0.6	0.597	0.941	0.939	0.940

Brun and Ernst [8] use a classification algorithm to find fault invariants which lead developers to hidden code errors. They reported classification precision (fixed relevance) for C of 45% and Java of 59% on average [8]. Ostrand et al. identified the top 20% of problematic files in a project using bug predictors and a negative binomial linear regression model [50, 51]. Using the top 20% of files from this list, they predict 71-93% of future bugs. Khoshgoftaar and Allen have proposed a model to list modules according to software quality factors such as future bug density [27, 28]. Their results showed that the top 10% of the identified modules have 64% of all bugs, and the top 20% have 82% of all bugs.

The change classification approach can predict bugs with 62% to 92% accuracy at the file change level of granularity. With a file-level change having, on average, 20 LOC, this is the smallest prediction granularity in the literature. In addition, this approach does not require any pre-identified problem lists to predict bugs. Overall, the combined prediction accuracy and granularity exceed the state of the art reported in the literature.

5.6.2 Recall-precision Curve

There are tradeoffs between precision and recall, and it is often possible to improve recall by reducing precision and vice versa. The best tradeoff between buggy change recall and precision remains an open question. Buggy change recall can easily go up to 100% by predicting all changes as buggy, but the precision will be very low. The recall-precision curve shows the trade-offs between recall and precision.

Figure 5-4 shows the recall-precision curves of 4 selected projects, Apache, Bugzilla, Mozilla, and Scarab. The curve for the Apache project shows that the precision grows up to about 60% (with 10% recall). For Mozilla and Scarab, the precision can reach 85-90% by lowering the recall to 30%.

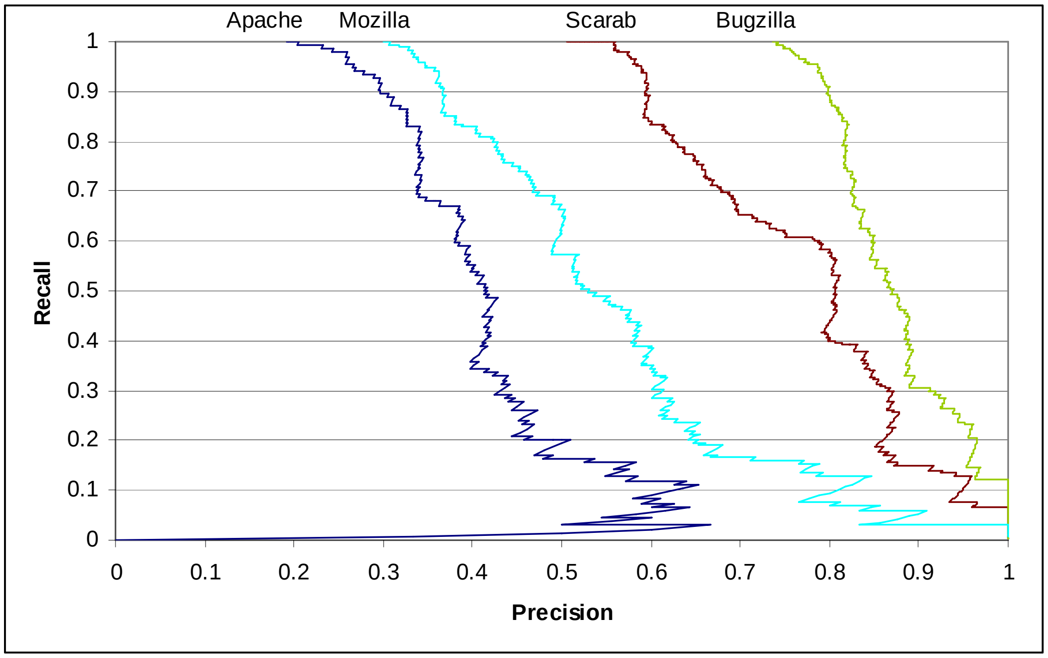

Figure 5-4. Recall-precision curves of selected 4 projects, Apache, Bugzilla, Mozilla, and Scarab using SVM.

Most machine learning classifiers use a threshold value to classify instances. For example, SVM uses the distance between each instance and the hyperplane to measure the weights of each instance. If an instance's weight is greater than the threshold value, the instance belongs to class1, otherwise it belongs to class2. By lowering or raising the threshold, it is possible to change recall and precision.

Usually by lowering recall, precision can be increased. However, sometimes lowering recall does not raise precision. For example, by lowering recall and changing the threshold value, correctly predicted buggy changes can be lost, thereby lowering precision.

Suppose there are change instances ordered by their weights in a one dimensional space as shown in Figure 5-5. An instance whose weight is smaller than a threshold value (to the left of the threshold) is classified as buggy. By selecting threshold 4 in Figure 5-5, the buggy change recall is 100% and buggy change precision is 50%. By selecting threshold 3, recall is 67% (4/6) and precision is 67% (4/6). By selecting threshold 2, recall is 50% (3/6) and precision is 75% (3/4). As this example demonstrates, when recall is reduced, precision usually increases. However, by selecting threshold 1, the recall is 17% (1/6) and precision is 50% (1/2) which is lower than the precision with threshold 2.

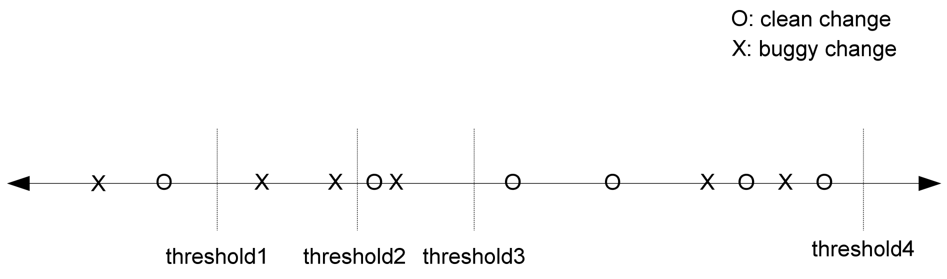

Figure 5-5. Sample instances in a one dimensional space.

To see how the change instances in the real projects are ordered, and why there is jitter in the recall-precision curves in Figure 5-4, the buggy and clean change counts are listed in Table 5-6. First, all instances are ordered by their SVM weights (the left side for buggy and right side for clean, like instances in Figure 5-5). Then the ordered instances are divided into 10 intervals, and buggy and clean instances in each interval are counted. For example, in the Scarab project the leftmost 76 instances include 65 buggy and 11 clean changes. The rightmost 72 instances include 70 clean and 2 buggy changes. For the Apache project, even the leftmost instances include many clean changes. The many clean changes on the left side cause jitter in the recall-precision curve, and make it impossible to attain 100% precision.

Table 5-6. Buggy and clean counts of ordered change instances by SVM weights.

A1	Total	70	70	70	70	70	70	70	70	70	70
	Buggy	32	27	21	15	19	10	4	4	1	1
	Clean	38	43	49	55	51	60	66	66	69	69
SCA	Total	76	72	72	72	72	72	72	72	72	72
	Buggy	65	60	52	46	29	30	23	38	21	2
	Clean	11	12	20	26	43	42	49	34	51	70

How are the recall-precision curves obtained using the 10-fold cross-validation approach? For each fold, the change instance SVM weights of the test set are computed. After applying 10-fold cross validation, all instance weights are merged and ordered to determine recall-precision rates by changing the threshold. In fact, 10 different classifiers are trained to create 10 models, and all 10 models are used to compute instance weights. 10 recall-precision curves are computed from 10 different classification models.

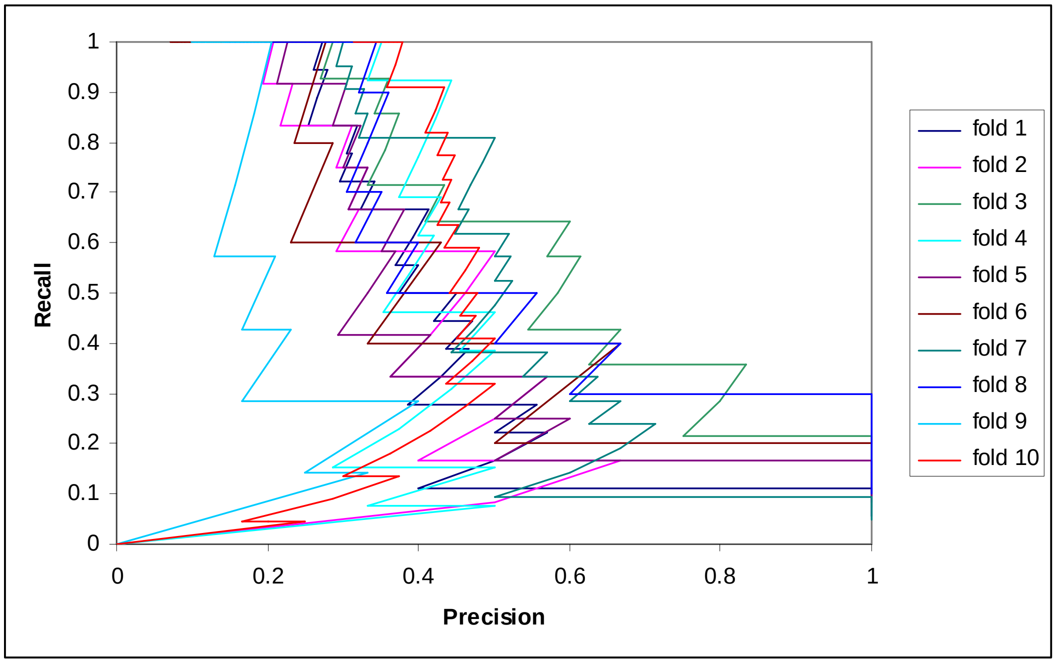

Figure 5-6. Recall-precision curves of each fold in the Apache 10-fold cross validation.

For example, Figure 5-6 shows the precision-recall curves of the 10 classification models used in the 10-fold cross validation of the Apache project. Each recall-precision curve is slightly different, but they share curve trends. However, to simplify the recall-precision curves, the mixed weights from 10 classification models are used to show the recall-precision curves in Figure 5-4.

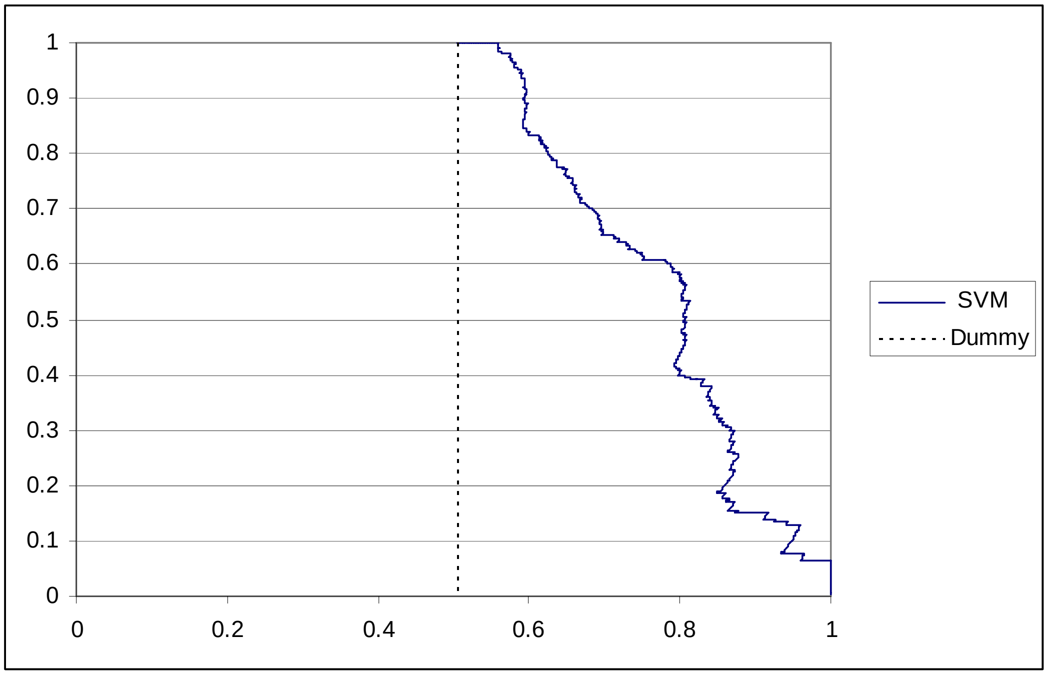

Figure 5-7. Recall-precision curves of SVM and dummy classifiers for the Scarab project.

How is this recall-precision better than other approaches, such as randomly guessing changes (dummy classifier) as buggy or clean? Since there are only two classes, it can be assumed that the chance of correct prediction is about 50%. For example, in the Scarab project, 50.5% of changes are buggy changes (See Table 5-3), so about 50% of the random guesses will be correct. Is this better than the results when using SVM? The recall-precision curves of the dummy and SVM classifiers for the Scarab project are compared in Figure 5-7. The dummy classifier's precision is stuck at 50.5%, while the SVM precision grows up to 85% (with 30% recall). SVM can improve buggy change precision by 35% in the Scarab project.

5.6.3 Feature Groups

The accuracy of the different feature group combinations discussed in Section 5.4.1 was observed. First, a classification model is trained using features from one feature group. Then a classification model is trained using all feature groups except the one feature group. In addition, the combination of features extracted solely from the source code (added delta, new revision source code, and deleted delta) were examined. Figure 5-8 shows the change classification accuracy for the Mozilla and Eclipse projects using various feature group combinations. The abbreviations for each feature group are shown in Table 5-4. The '~' mark indicates that the corresponding feature group is excluded. For example, '~D' means all features except for D (Deleted delta) are used. The feature group "AND" is the combination of all source code feature groups (A, N, and D). The accuracy trend of the two projects is different, but they share some properties. For example, the accuracy obtained by using only one feature group is lower than using from multiple feature groups.

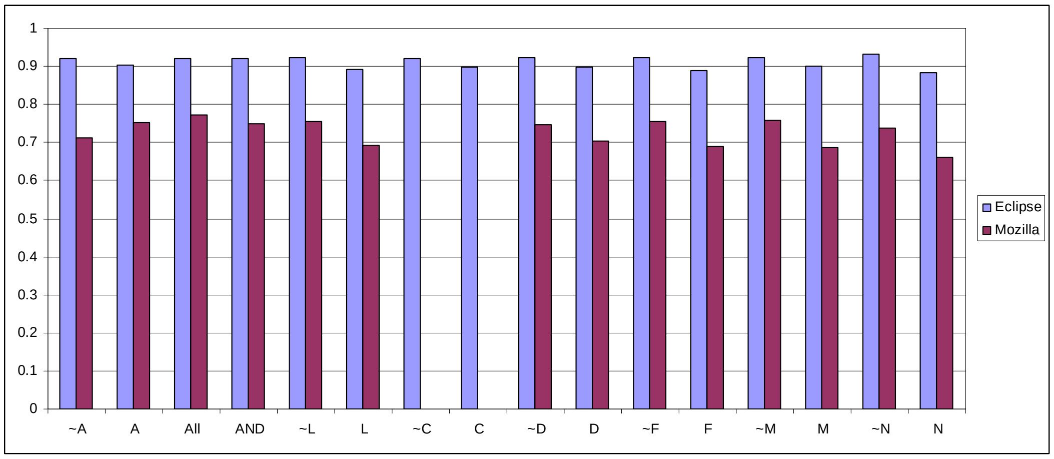

Figure 5-8. Feature group combination accuracy of Eclipse and Mozilla using SVM. Note that complexity metrics (C) for Mozilla are not available so ~C and C for Mozilla are shown as 0.

The average accuracy of 12 open source projects using various feature combinations is shown in Figure 5-9. Using a feature combination of only source code (A, N, and D combined) leads to a relatively high accuracy, while using only one feature group from the source code, such as A, N, or D, does not lead to high accuracy. Using only 'L' (change log) leads to the worst accuracy. This is a somewhat surprising result, since only the change log text has human readable semantics. The rationale behind this phenomenon is explored further in the discussion chapter.

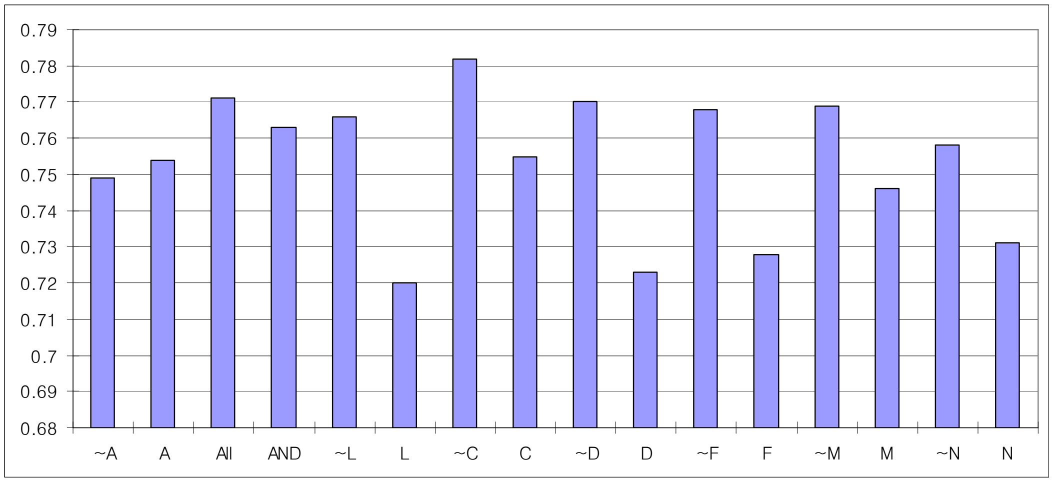

Figure 5-9. Average feature group combination accuracy across 12 projects using SVM

After analyzing the combinations of feature groups, the feature combination that yields the best accuracy and best recall for each project is identified, as shown in Figure 5-10 and Figure 5-11. The results indicate that there is no feature combination that works best across all projects. To achieve the best prediction accuracy, each project requires a project-specific feature selection process.

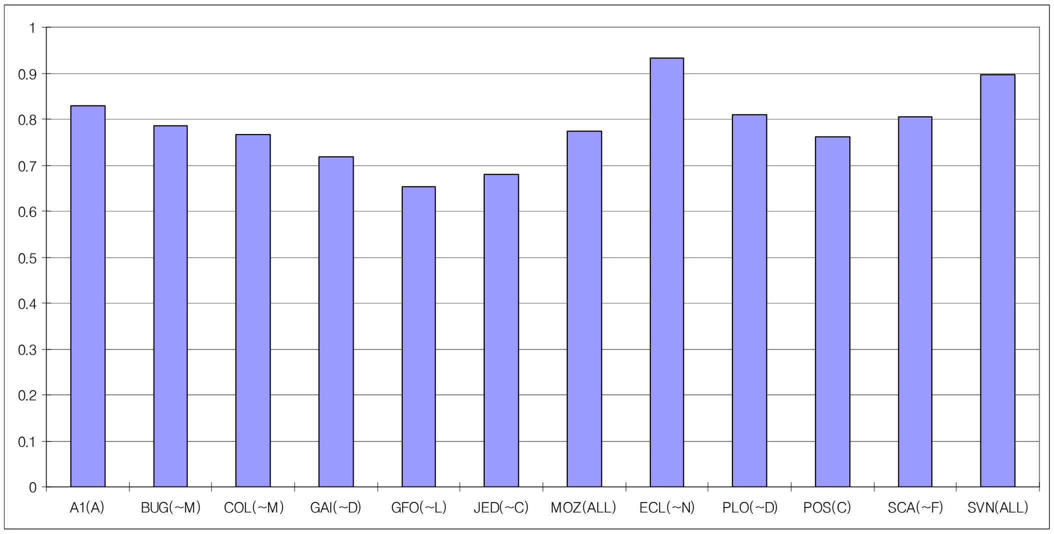

Figure 5-10. Feature group combination yielding best classification accuracy using SVM

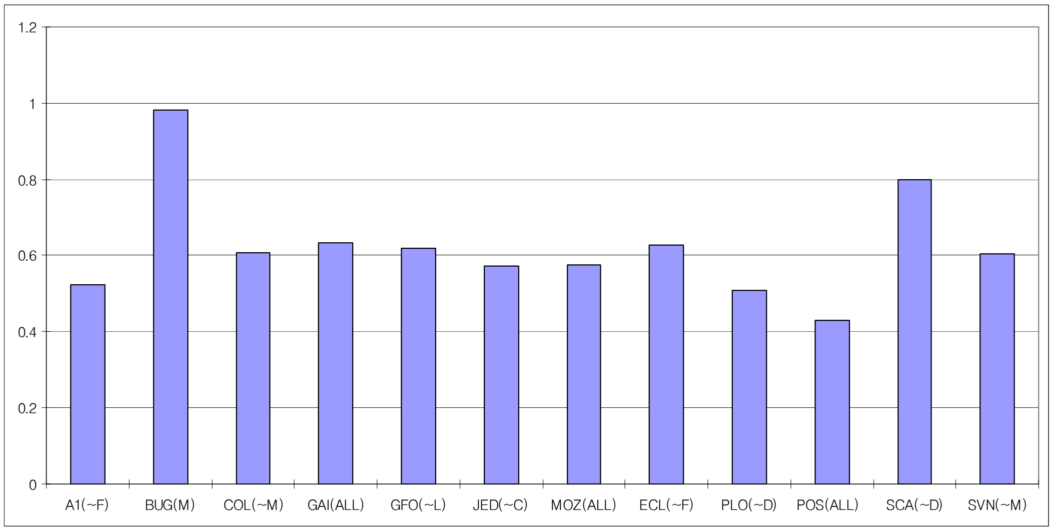

Figure 5-11. Feature group combination yielding best buggy change classification recall using SVM

5.6.4 Important Feature Groups

It is necessary to understand the importance of feature groups for classification, since this affects feature selection and further feature creation. To find important feature groups, each individual feature was ranked and the number of group features ranked in the top 10% was counted. Let ***F*** be the set of all features, and F_i be the *i*th feature group: $F_i = \{f_{i1}, f_{i2}, \ldots, f_{in}\}$, and $F = \bigcup_{i}^{n} F_i$.

All features in *F* were sorted by their chi-square rank [9]. Note that chi-square [9] computes rank of each feature individually. Usually machine learning classifiers use combined features. The ranks from chi-square [9] might not the same as the importance of features in a machine learning classifier. However, chi-square rank [9] give us a general idea of the importance of each feature.

Let F_{top} be the set of features ranked in the top 10%. Then the top feature sets, F_{topi} that contained these features originally in F_i is obtained.

$$F_{top_i} = \{ f_{in} \mid f_{in} \in F_{top} \}.$$

The importance of feature group *i* is defined by

$$F_{imp_i} = 10 \frac{|F_{top_i}|}{|F_i|}$$

The importance value ranges from 0 to 10. Larger importance values indicate a more important feature group.

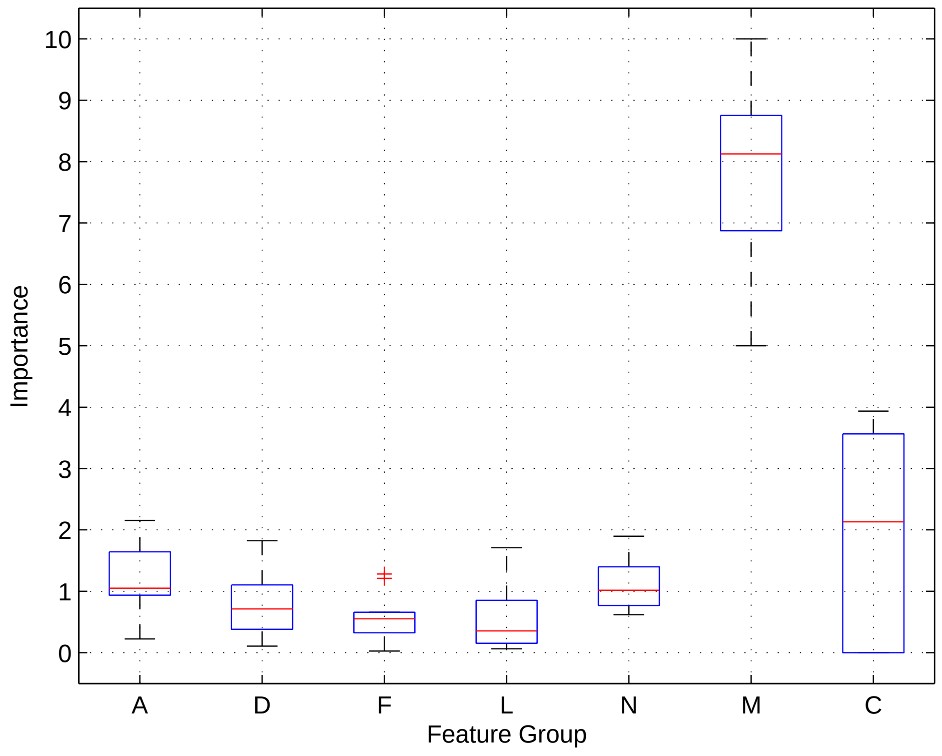

Figure 5-12. Importance of each feature group.

Figure 5-12 shows the importance of each group using a box-plot. The importance of each feature for 12 open projects is computed, and the average, 25% quartile (bottom of box) and 75% quartile (top of box) are shown. For example, features in ‘M’ (Metadata) are important features, since the average of the importance is 8.1. ‘F’ (file name) and ‘L’ (new revision source code) are less important features. The importance of ‘C’ (complexity metrics) varies from project to project, since the distance between the 25% and 75% quartiles is long.

5.6.5 Important Individual Features

The important individual features were highlighted from each feature group by ranking individual features using a chi-square metric [9]. After ranking individual features, the distributions of each feature in the buggy and clean changes were identified based on whether the corresponding feature contributed to either buggy or clean changes.

The top 5 ranked individual features within each feature group are listed in Table 5-7 with the overall rank of the feature in the parenthesis and an indication of whether it is contributing to the buggy or clean change class (+ for buggy and – for clean). The important features of each project vary due to the unique bug properties of each project.

Table 5-7. Top ranked individual features of each group. The '+' sign indicates the feature contributes to buggy changes. The '–' sign indicate the feature contributes to clean changes. The number in parentheses indicates the overall rank of the feature. The △ mark beside a complexity metric indicates it is a delta metric.

	Apache	Bugzilla	Eclipse
Complexity metrics	△CountLine (+4), CountLineCode (+7), CountStmt (+8), △SumCyclomaticStrict (+9), CountStmtExe(+11)	N/A	SumEssential(-117), △CountLineBlank(+228), CountStmtDecl(-417), CountLineComment(-419), CountLineCodeDecl(-420)
Change Log	and(+196), http(+356), copyright (-395), with(+408), via(+636)	fix(+345), comments(-351), correcting (-414), patch(+480), ability(- 492)	fix(+386), for(+398), 18(+961), 3249(+962), 1(-1795)
Metadata	changed_loc(+1), bug_count(+38), time (+156) changed_log_length (+216) , author(+512)	changed loc(+1), loc(+2), bug count(+3), time(-9), change count(+43)	time(+74), changed loc(+88), bug count(+104), days(-137), change log length(-142)
New Source	current(+10) , step(+23), variable(+27), additionally(+29), false (+30)	order(+4), b(+7), bit(+8), ok(+10), used(+11)	flowinfo(+1), analysecode(+2), flowcontext(+3), slow(+4), iabstractsynt(+6)
Added Delta	if(+2), is(+3), 0(+5), else(+6), <(+15)	if(+5), my(+6), value(+15), not(+22), sendsql(+26)	codestream(+12), recordpositionsfrom(+8), belongsto(+24), complete(+25), jobfamily(+26)
Deleted Delta	if(+18), ==(+21), of(+33), int(+43), else(+53)	name(+153), value(+281), my(+300), fetchsqldata(+326), sendsql(+349)	codestream(+14), public(+15), recordpositionsfrom(+19), return(+21), this(+22)
Directory/File name	protocol(+372), main(+468), include(-702), rewrite(+ 841), c(+1171)	relation(-219), set(-220), move (-375), createattachment (-490), export(-491)	ast(+5), compiler(+47), statement(+108), core(- 116), model(-214)

In the Apache project the most important feature by chi-square [9] is '*changed_loc*' in the Metadata feature group. It indicates the more lines change, the greater is the risk of a change being bug-introducing. The second most important feature in the Apache project is the "*if*" keyword in the added delta. It indicates that adding a new *if* condition is a risky change. This rank is computed for each feature individually and the listed features are only a small portion of the entire set of features. Further analysis of the top ranking features may lead to better understanding of the causality of bug introduction or permit analysis of common buggy change patterns. This remains future work.

5.7 Discussion

This chapter discusses possible applications of change classification, and provides a more detailed analysis of the rationale of the results. Two results of different classification algorithms are compared. This chapter ends with some discussion on the limitations of the change classification experiment.

5.7.1 Applications

Right now, the buggy change classifier operates in a lab environment; however, it can be put into use in various ways:

- **A commit checker**: The classifier identifies buggy changes during commits of changes to an SCM system, and notifies developers of the results. Bug prediction in the commit checker is immediate, thus making it easy for developers to inspect the changes they have just made.
- **Potential bug indicator during source code editing**: The results show that the features extracted from source code (groups A, N, D) have a strong discriminative power. Just using features from source code, it is possible to perform accurate bug classification. This implies that a bug classifier can be embedded in a source code editor. During the source code editing process, the classifier could monitor source code changes. As soon as the cumulative set of changes made during an editing session leads the classifier to make a

bug prediction, the editor can notify the developer.

- **Impact on the software development process**: Results from the change classifier could be integrated into the software development process. After committing a change, a developer can receive feedback from the classifier. If the classifier indicates a buggy change, this could trigger an automatic code inspection on the specified change. After the inspection, the developer commits a modified change and receives more feedback. Using the classifier in this way would lead to a new low-level development cycle: make a change, receive feedback, inspect the change, and repeat.

5.7.2 SVM and Naïve Bayes

SVM is one of the most effective machine learning algorithms for text classification [26]. This section compares the accuracy of SVM and Naïve Bayes. For each classifier, trained and validated using the corpus described in Table 5-3, the accuracy is shown in Figure 5-13. SVM performs better for most projects. In some projects such as Gaim, JEdit, and PostgreSQL, Naïve Bayes performs as well as or better than SVM. These results indicate that different projects have different buggy/clean change patterns, and hence the best classification algorithm for one project is not necessarily the best classifier for another project.

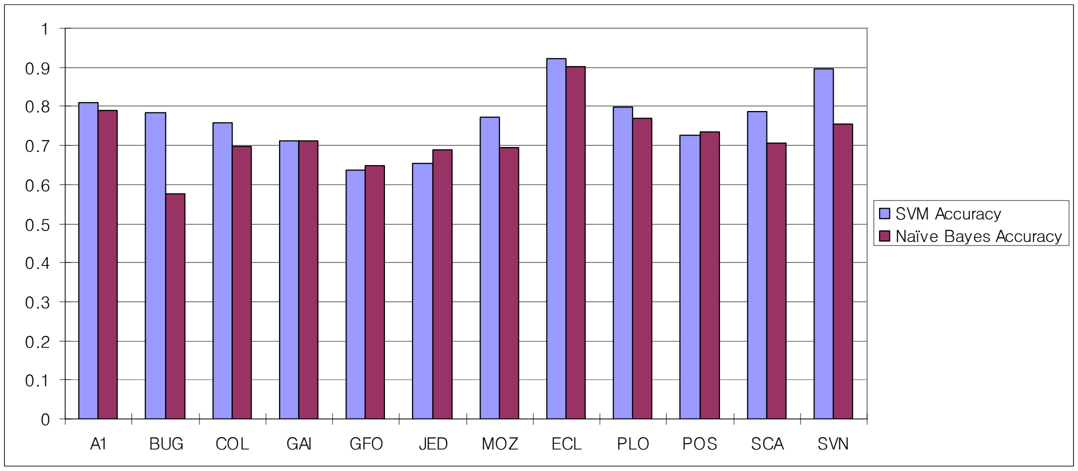

Figure 5-13. SVM and Naive Bayes Accuracy comparison

5.7.3 Project Periods and Accuracy

For this project, the corpus (as shown in Table 5-3) uses file changes from revisions 500-1000 (500-750 for bigger projects) of each project. The first part of a project history may have noise in change patterns. A project is typically unsettled early on, with many refactorings and unusual change activities. Changes in the later periods are used on the assumption that these later periods have change patterns that are more regular.

This section tests this assumption by comparing the classification results of the first and second periods of each project history. A second corpus was made for revisions 1-500 (1-250 for bigger projects) using the same methods used in creating the corpus for revisions 500-100 (500-750 for bigger projects). Figure 5-14 compares the prediction accuracy for the two periods of all 12 projects. The accuracy for the second period (500-1000) is generally higher than that of the first period. However, in some projects the accuracy of the first period is higher than that of the second period. Overall, the prediction accuracy of the first and second periods is similar. This is encouraging, since it shows that bug classification can be used relatively early in a project's lifecycle. It indicates the assumption of the first period in history [23] is wrong for the change classification, and change classification can be used for periods that may have many refactorings or unusual change behaviors.

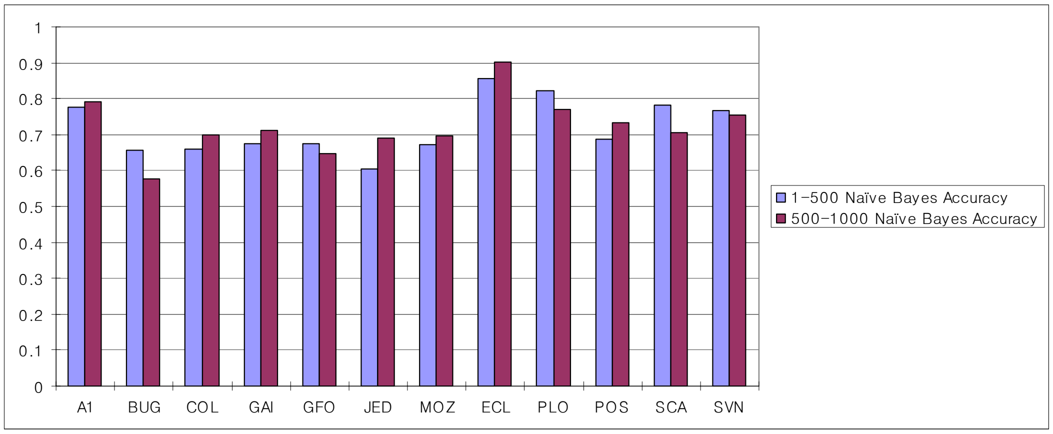

Figure 5-14. Accuracy of the revision 1-500 and revision 500-1000 using Naïve Bayes.

5.7.4 Classification Granularity

One of the advantages of classifying file changes is that it provides predictions at a small level of granularity (a single change to a single file). Table 5-8 shows the average LOC in a file change, the LOC in the file, entity (such as functions or methods) changed LOC, and entity LOC of the 12 projects. For example, the average LOC for file changes in Apache 1.3 is 15.42, and average LOC of files in Apache 1.3 is 455.73. The average LOC for file change of all projects is 20, while the average LOC for files is 300, including comments, whitespaces, and blank lines. For example, if a tool predicts bugs at the file level in the corpus used in this study, it is necessary to inspect 300 LOC on average to locate the line(s) containing the bug. Since the change classification approach classifies file changes, the prediction is at the file change level, and hence only 20 lines on average need to be inspected.

Table 5-8. Average LOC. Entity and entity change LOCs are available for only projects written C/C++ and Java.

Project	File change	File	Entity change	Entity
Apache HTTP 1.3 (A1)	15.42	455.73	15.83	28.32
Bugzilla (BU)	18.30	375.37	N/A	N/A
Columba (CO)	14.94	143.3	10.99	15.64
Gaim(GA)	19.64	832	11.09	38.43
GForge(GFO)	17.73	155.49	N/A	N/A
JEdit (JED)	23.64	325.78	7.65	18.74
Mozilla (MOZ)	21.20	285	N/A	N/A
Eclipse(ECL)	48.26	230.29	13.77	16.9
Plone(PLO)	9.7	49.11	N/A	N/A
PostgreSQL (POS)	14.28	282.92	25	32.21
Scarab (SCA)	21.75	145.98	15.21	16.06
Subversion (SVN)	15.35	354.31	11.923	33.81
Average	20.02	302.94	13.93	25.01

It may also be possible to locate bugs at an even finer granularity than file changes. Using the same techniques described in this dissertation, it is possible to classify entity (function or method) changes (14 LOC per change) instead of file changes. Furthermore, it is possible to analyze the contents of each line in the buggy changes, and ignore comments or blank lines to provide even finer granularity for the prediction. This remains future work.

5.7.5 Discriminative Power of Change Log Features

The change log for a project contains a human readable description of each change, and it is assumed to be a significant feature. The features from change logs are not significant features as shown in Figure 5-9. The low discriminative power of change log features is due to the high number of file changes in each commit.

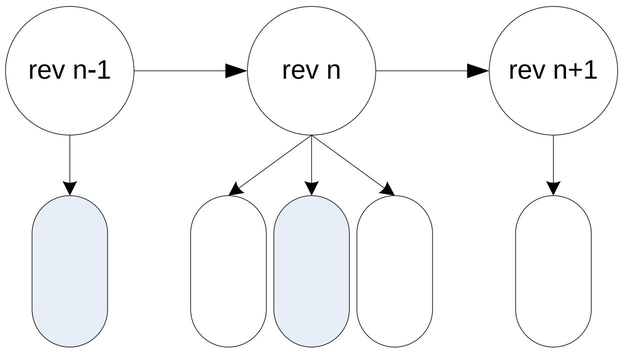

Figure 5-15. Example commits and corresponding number of file changes

Suppose we have commits and file changes as shown in Figure 5-15. In revision *n-1* and revision *n+1*, there is only one file change in each commit. The commit at revision *n* has three file changes. Assume that the gray colored file changes are buggy and white colored file changes are clean. If there is only one change per revision, the file change and change log have a one-to-one mapping. However, if there is more than one file change, the file changes and change log are a many-to-one mapping. When multiple file changes in the same commit are different change types, such as two clean changes and one buggy change, as shown in Figure 5-15 at revision *n*, it decreases the discriminative power of features extracted from change log messages.

If there are many file changes in a commit, it is likely to have a mixture of change types across these files. Change log features have more discriminative power when the file change types in the same commit are consistent and/or each revision has a small number of file changes.

5.7.6 Correlation between Percentage of Bug-introducing Changes and Classification Accuracy

One observation that can be made from Table 5-3 is that the percentage of changes that are buggy varies substantially among projects, ranging from 10.1% of changes for Eclipse to 73.7% for Bugzilla. One explanation for this variance is the varying use of change log messages among projects. Bugzilla and Scarab, being change tracking tool projects, have a higher overall use of change tracking. It is likely that for those projects, the class of buggy changes also encompasses other kinds of modifications. For these projects, change classification can be viewed as successfully predicting the kinds of changes that result in change tracking tool entries.

One question that arises is whether the percentage of buggy changes for a project affects change classification performance such as accuracy, recall, and precision? A Pearson correlation was computed between the percentage of buggy changes, and the measures of accuracy, recall, and precision for the 12 projects analyzed in this paper. Figure 5-9 lists the correlation values. A correlation value of 1 indicates tight correlation, while .5 indicates almost no correlation. The values show no correlation for accuracy, and weak but not significant correlations for buggy recall and precision.

Table 5-9. Correlation between the percentages of buggy changes and change classification performance.

	Buggy % vs. accuracy	Buggy % vs. bug recall	Buggy % vs. bug precision
Correlation	-0.56	0.77	0.64

5.8 Summary

If we know that a change we just made contains bugs, it will help us to identify and fix the potential bugs in the change before a bug report. Experimental results presented in this chapter show that it is possible to classify buggy changes with acceptable accuracy (78% on average) and buggy change recall (65% on average) using features from change information. Developers can benefit from focused and prompt prediction of buggy changes, receiving this prediction either while they are editing source code or right after a change submission.

It is the first research to classify file changes using the combination of change information features and all source code terms.

5.9 Future Work

Although these experimental results are encouraging, there is still room for improvement. The future work will include the following topics:

Exploring on-line machine learning algorithms to learn and update a classification model as the project progresses and using it to predict future changes. After randomizing change instances, the ten-fold cross validation is used to validate the change classification approach. In practice, it is necessary to develop on-line machine learning algorithms that gather instances from revision *1* to *n* and trains a classifier to predict revision *n+1* changes.

The change or bug patterns may differ from developer to developer. If each developer's buggy and clean changes are used to train a developer specific classifier, it may lead to more accurate change classification. A developer-specific classifier can be used to classify the corresponding developer's changes. Training developer-specific change classifiers and applying them to each developer's changes is also future work.

Even though many features are used to classify changes in this dissertation, generating more features from change information and exploring various ways to extract features such as latent semantic analysis [36] may lead to more accurate classification.

In this study, two machine learning algorithms, Naïve Bayes and SVM are used. It may possible that other machine learning algorithms yield better accuracy, recall, and precision. Classifying changes with other machine learning algorithms such as decision trees and neural networks is also future work.

Modifying existing machine learning algorithms to achieve better prediction accuracy, precision, and recall [16] is also necessary. Since the source code change classification is different from regular text classification, modified machine learning algorithms may work better than existing ones.

6 Threats to Validity

There are four major threats to the validity of this work.

Systems examined might not be representative. In this dissertation 12 systems are examined, more than any other work reported in the literature. In spite of this, it is still possible that systems that have better (or worse) bug classification accuracy or bug cache hit rates than a random selection of systems. Since systems were only chosen that had some degree of linkage between change tracking systems and the text in the change log (so it is possible determine fix inducing changes), there is a project selection bias. It certainly would be nice to have a larger dataset.

Systems are all open source. The systems examined in this dissertation all use an open source development methodology, and hence might not be representative of all development contexts. It is possible that the stronger deadline pressure, different personnel turnover patterns, and different development processes used in commercial development could lead to different buggy change patterns or bug localities.

Bug-fix data is incomplete. Even though projects that have good quality change logs were selected, the heuristic bug-fix change identification method is still 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 heuristic method relies on change logs to identify fixes, the identified fixes may not all be true fixes.

Bug-introducing change data is incomplete. Since the bug-introducing change identification algorithm uses hunks in the fix changes [32], the algorithm may miss certain types of bug-introducing changes such as only addition fix changes that only involve addition of text or bug-introducing changes caused by other parts of the source code.

7 Related Work

Work related to this thesis falls into one of the followings types: bug prediction, source code classification and text classification. This chapter describes them and compares them with the bug cache and change classification approaches.

7.1 Bug Prediction

There is a rich literature for bug detection and prediction. This work falls into the following three categories: *identifying a problematic module list* by applying software quality metrics or change history [23, 27, 28, 35, 50, 51], *predicting the bug density* of each module using its software change history [20, 46], and *detecting bugs by analyzing source or binary code* using static or dynamic analysis techniques, including type checking, deadlock detection, automatic theorem proving, or pattern reorganization [14, 24, 38, 60]. This chapter discusses the first two categories, since they are related to this dissertation. Additionally, the bug prediction algorithms that use project histories are discussed.

7.1.1 Identifying Problematic Entities

If it is possible to identify problematic software entities in advance, developers can use caution when changing them or can run intensive test cases on them. Classification or regression algorithms using various features such as complexity metrics, cumulative change count, or bug count are widely used to predict problematic entities. Table 7-1 summarizes selected bug prediction approaches by identifying problematic entities.

Table 7-1. Selected bug prediction research.

Authors	Approaches	Analyzed projects	Prediction granularity	Performance
Gyimothy et al. [22]	Decision trees and neural networks with object-oriented metrics.	Mozilla	Class (File)	Recall/precision is 70%
Brun et al. [8]	SVM and decision trees to find relevant program properties (fault invariant). The fault invariant is used to find errors	Get, Pathfinder, etc.	Invariants	Fixed relevant: 45% for C and 59% for Java
Hassan et al. [23]	Select top-ten modules using four factors separately: most frequently modified, most recently modified, most frequently fixed, and most recently fixed	OpenBSD, NetBSD, FreeBDS, KDE, Open office	Module (directory)	The top-ten list finds 50% of bugs
Ostrand et al. [50, 51]	A negative binomial regression model using fault and change history	Two industrial system	File	Using 20% of file list, the accuracy is 71-93%
Khoshgoftaar et al. [27, 28]	A step wise regression with quality factor such as the number of faults and software complexity metrics	Two industrial system	Module	Using 10% of the modules, the accuracy is 64% (82% using 20% of modules)

Gyimothy et al. used regression and machine learning algorithms such as decision trees [2] and neural networks [2] to predict fault classes of the Mozilla project in each release (releases 1.0~1.6) [22]. They use object-oriented metrics as features for the machine learning algorithms. Their recall and precision are about 70% while the change classification accuracy in this study for the Mozilla project is 77.3% and precision is 63.4%. Note that Gyimothy et al. predicted faults in the class level, while the prediction of the change classification approach is at the granularity of file changes. Gyimothy et al. used release-based classes for prediction, and a release is an accumulation of many revisions.

Brun and Ernst [8] use two classification algorithms to find hidden code errors. Using Ernst's Daikon dynamic invariant detector, invariant features are extracted from code with known errors and with errors removed. They train a Support Vector Machine and a decision tree using the extracted features, then classify invariants in the source code as either fault-invariant or non-fault-invariant. The fault-invariant information is used to find hidden errors in the source code. Reported classification accuracy is 10.6% on average (9% for C and 12.2% for Java), with classification precision of 21.6% on average (10% for C and 33.2% for Java), and the best classification precision (with top 80 relevant invariants) of 52% on average (45% for C and 59% for Java). The classified fault invariants guide developers to find hidden errors. Brun and Ernst's approach is similar to our work in that they try to capture properties of buggy code and use it to train machine learning classifiers to make future predictions. However, they used only invariant information as code properties, which leads to lower accuracy and precision. In contrast, change classification uses a broader set of features including source code, complexity metrics, and change metadata.

Hassan and Holt proposed a caching algorithm to compute the set of fault prone modules, called the top-ten list [23]. They used four factors to determine this list: software that was most frequently modified (MFM), most recently modified (MRM), most frequently fixed (MFF), and most recently fixed (MRF). This work is similar to ours in that they dynamically maintain the list using the current status of the project.

The analysis in [23] uses the four factors separately to compute the list while the bug cache approach uses all factors together to derive synergy from all possible factors. For example, the bug cache approach uses MRF when the cache fetches a missed entity, and MFF if the cache replacement algorithm is BUG. The bug cache uses MRM when the cache pre-fetches a changed entity. The bug cache additionally uses co-change measurement (spatial locality) as a predictor, which boosts the performance of the bug cache approach.

The granularity of the top-ten list in [23] is very large, at the subsystem level, where a subsystem is a collection of files. The entire bug caches operate at smaller granularity, which is more useful since it allows developers and testers to focus on a smaller set of code.

Similar to the top-ten list, Ostrand et al. identified the top 20% of problematic files in a project [50, 51]. Using future fault predictors and a negative binomial linear regression model, they predict the fault density of each file. These files are then ordered by fault density. Using the top 20% of files from this list, they predicted 71-93% of future faults. This is most directly comparable to Figure 4-9 (bug cache using fixes), where the bug cache can predict 73-96% of future faults, but with the bug cache achieving greater precision (10% of files for bug cache vs. 20% for Ostrand et al.).

Khoshgoftaar and Allen [27, 28] propose a model to list modules according to software quality factors such as fault density or software complexity metrics such as LOC, number of unique operators, and cyclomatic complexity. A step-wise regression is then performed to find weights for each factor [27, 28]. Their results show that the top 10% of the identified modules have 64% of all identified faults, and the top 20% have 82% of all faults. Since [27, 28] use software complexity metrics, the list of buggy modules tends to be static over time, and hence does not easily adapt to changing observed bug densities. Finding a bug in a module does not change its rank, since it does not change the software's computed complexity.

Most existing research focuses on classifying or predicting future bugs at the level of modules, files, or functions. The change classification approach classifies changes by learning from previous changes. Software complexity metrics and change measures are widely used features, but most researchers in the literature neglect terms in the source code when they extract features from the project history. The change classification approach uses every term in the source code including operators, and shows that such features have a strong discriminative power.

7.1.2 Predicting Fault Density

Graves et al. compute the fault density of each software module using a weighted time damp model that uses changes over time as a future fault prediction factor [20]. They assume that a module with old changes is either a fixed module or a fault-free module, and weight recent changes over older ones. They observed a significant improvement in predictive accuracy with this approach, providing additional support for the locality of bugs to changes.

Mockus et al. identify significant change types such as the number of co-changed subsystems, the number of change deltas, and whether the change was a fix, or change frequency [43]. They used logistic regression [2] for their prediction model. They identified significant factors such as whether the change was a fix, and their change interval was similar to the temporal bug locality concept used in the bug cache. The co-changing subsystem numbers are also similar to the spatial bug locality concept used in this study [43].

Nagappan et al. indicate that relative code change measurements are a better fault predictor that absolute change measurements [46]. For example, instead of using absolute changed-LOC as a predictor, they use changed-LOC/LOC instead. The bug cache approach uses absolute bug and change measurements to fetch or replace entities. The granularity of Nagappan's prediction in [46] is at the entire binary level, and hence it is unclear how well their approach works on smaller grain sizes.

7.1.3 Using Project History

This section surveys research that uses project histories, including building project knowledge [10, 11], detecting common bug patterns [37, 64], and finding association rules among bugs. [59].

Hipikat is a tool that recommends relevant software artifacts to the current developers' task based on project histories comprised of artifacts such as source code changes, mailing list messages, bug tracking entries, and written documentation [10, 11]. The Hipikat approach is similar to the work in this dissertation in that it uses project history to learn project specific knowledge. However, bug cache and change classification approaches explicitly identify bad (bug) and good (fix) changes to detect potential bugs. Hipikat tries to provide related references to developers rather than identify good or bad changes.

Williams and Hollingsworth used project histories to improve existing bug finding tools [64]. It is very common to use a call function and its return value. Performing a null check on a return value can validate its correctness. If left unchecked, the return value may contribute to the generation of a bug. The problem in this approach is that there are too many false positives. The bug finding tool generates warnings about all source code that uses an unchecked return value, thereby causing a high false positive rate. To remove these false positives, Williams and Hollingsworth use project histories to determine what kinds of function return values must be checked. For example, if the return value of the function '*foo*' was always checked in the project history, but not checked in current source code, it is very suspicious.

Livshits and Zimmermann combined software repository mining and dynamic analysis to discover common method usage patterns that are likely to encounter violations in Java applications [37]. Their approach employs dynamic analysis and is more specific in finding violation patterns on method usage pairs, for example *blockSignal()* and *unblockSingal()* should exist in a pair in the source code.

Song et al. found association rules among six bug types from project histories [59]. Using these association rules, they can predict future bugs. For example, suppose bug types A and B are often found together in the history. Then if we find only bug type A in source code, we assume the code contains bug type B as well.

7.2 Source Code Classification, Clustering, and Associating

Source code features (terms) have also been used in software classification [33], clustering [34, 39] and associating source code to other artifacts such as design documents [40]. Krovtez et al. used terms in the source code (as features) and SVM to classify software projects into broad functional categories such as communications, databases, games, and math [33]. Their insight is that software projects in the same category will share terms in their source code, thereby permitting classification. Maletic et al. used all terms in the source via Latent Semantic Analysis (LSA) to cluster software and associate other relevant software project documents [39, 40]. Kuhn et al. used partial terms from source code to cluster the code to detect abnormal module structures [34].

Research that categorizes or associates source code with other documents is similar to ours in that it uses terms from the source code. However, the change classification approach uses features from multiple sources such as change deltas, source code, change log text, and change metadata rather than using only the source code. The goal of associating source code with other lifecycle documents differs from this study, since this study tries to identify buggy changes, while this other work associates relevant artifacts or tries to find abnormalities of module structures.

7.3 Text Classification

Text classification is a well-studied area with a long research history. Using text terms as features, researchers propose algorithms to classify text documents [57], such as classifying news articles into their corresponding genres. Among existing work on text classification, spam filtering [67] is the most similar to ours. Spam filtering is a binary classification problem to identify email as spam or ham (not spam). The change classification approach adapts existing text classification algorithms into the domain of source code change classification. My research focuses on generating and selecting features related to buggy source code changes.

8 Conclusion

This dissertation presented two adaptive bug prediction approaches that fully leverage project history. The bug cache approach predicts 73-95% buggy entities using a cache with 10% of all the files in the software. The change classification approach classifies changes with 75-95% accuracy after training a classifier on a project's change history. The two approaches are project specific and maintain adaptive bug prediction models, meaning that the models learn from previous mistakes in change history. The two approaches yield reasonable bug prediction results that generally exceed the best results in the literature in either accuracy, granularity, or both. The two approaches show that the use of history and adaptive prediction models are effective for bug prediction.

The following are contributions of this dissertation:

Adaptive bug prediction approaches using project history: Two adaptive bug prediction approaches using project history are presented. In the literature, many static bug prediction models are widely used. The presented bug prediction models have a learning process so that the learned prediction models are project specific. The wide range of features from a project history, including keywords, changed delta, complexity metrics, file name, author, and change time are used to build project specific prediction models and predict future bugs. For example, the bug cache approach uses each prediction result to update the cache adaptively. The change classification approach uses features from each project to train a project specific classifier.

Leveraging bug-introducing change: A few researchers used bug-introducing changes to identify buggy source patterns [29] or found correlations between signature changes and buggy changes [31]. However, most bug prediction related research uses bug-fix data to find prediction models or to evaluate the models [23, 49, 51]. Bug-fix data can indicate where bugs are, but cannot indicate when and who created the bugs. The algorithm to identify bug-introducing changes was introduced by Śliwerski et al. [58] and improved by Kim et al. [32]. In this dissertation, bug-introducing change data is used, since determining when bugs would be introduced is more important than finding out when the bugs were fixed. Two approaches that were proposed in this dissertation can serve as references for research that uses bug-introducing changes.

Properties of bug occurrences: The experiments in this dissertation discovered that the occurrence of a bug was not in bursts, but rather local. The results of the bug cache clearly indicated that bug occurrences have strong temporal and special localities.

Bug cache algorithm that uses bug locality to predict future bugs: Based on the bug locality, this dissertation presented a simple algorithm, bug cache to predict future bug locations.

Combining prediction features together leads better accuracy: The bug cache approach combines many possible bug prediction features such as bug occurrence time, accumulated bug counts, co-change files, and change counts, and yields better accuracy that that of using a single features individually.

Software changes can be classified: Change classification is the first attempt to classify each software change as either buggy or clean using machine learning algorithms. This dissertation shows that source code is classifiable if the proper features are extracted and used.

Feature engineering for software change classification: To classify software changes, it is important to extract the appropriate features from software changes. This dissertation introduced feature engineering techniques from software changes. For example, feature extraction methods from all keywords in the source code, file names, and metadata are described.

Important features for change classification: Using the feature source combinations and chi square measurements, the important feature groups (Figure 5-10) and individual features (Table 5-7) are identified. The identified features could be used as good references for further research in selecting features, finding other feature sources, and developing classification algorithms.

Bug prediction approach using history: The two approaches in this dissertation showed that project history is a good data source for future bug prediction, since project history accumulates the bug and fix experiences of software projects.

Overall, I expect that future approaches will no longer see software history as a series of revisions and changes, but rather as a series of successes and failures—and as a source for continuous awareness and improvement. The bug cache and change classification approaches are the first steps in this direction.

Bibliography

[1] "Software bug," 2006, http://en.wikipedia.org/wiki/Computer_bug.

[2] E. Alpaydin, *Introduction to Machine Learning*: The MIT Press, 2004.

[3] B. Behlendorf, C. M. Pilato, G. Stein, K. Fogel, K. Hancock, and B. Collins-Sussman, "Subversion Project Homepage," 2006, http://subversion.tigris.org/.

[4] B. Berliner, "CVS II: Parallelizing Software Development," Proc. of Winter 1990 USENIX Conference, Washington, DC, pp. 341-351, 1990.

[5] J. Bevan and E. J. Whitehead, Jr., "Identification of Software Instabilities," Proc. of 10th Working Conference on Reverse Engineering (WCRE 2003), Victoria, Canada, pp. 134-145, 2003.

[6] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey, "Facilitating Software Evolution with Kenyon," Proc. of the 2005 European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, pp. 177-186, 2005.

[7] D. Beyer and A. Noack, "Clustering Software Artifacts Based on Frequent Common Changes," Proc. of the 13th IEEE International Workshop on Program Comprehension (IWPC 2005), St. Louis, Missouri, USA, pp. 259-268, 2005.

[8] Y. Brun and M. D. Ernst, "Finding Latent Code Errors via Machine Learning over Program Executions," Proc. of 26th International Conference on Software Engineering (ICSE 2004), Scotland, UK, pp. 480-490, 2004.

[9] J. Connor-Linton, "Chi Square Tutorial," 2005, http://www.georgetown.edu/faculty/ballc/webtools/web_chi_tut.html.

[10] D. Cubranic and G. C. Murphy, "Hipikat: Recommending pertinent software development artifacts," Proc. of 25th International Conference on Software Engineering (ICSE 2003), Portland, Oregon, pp. 408-418, 2003.

[11] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A Project Memory for Software Development," *IEEE Trans. Software Engineering*, vol. 31, no. 6, pp. 446-465, 2005.

[12] V. Dallmeier, P. Weißgerber, and T. Zimmermann, "APFEL: A Preprocessing Framework For Eclipse," 2005, http://www.st.cs.uni-sb.de/softevo/apfel/.

[13] M. Fischer, M. Pinzger, and H. Gall, "Populating a Release History Database from Version Control and Bug Tracking Systems," Proc. of 19th International Conference on Software Maintenance (ICSM 2003), Amsterdam, The Netherlands, pp. 23-32, 2003.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended Static Checking for Java," Proc. of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin, Germany, pp. 234-245, 2002.

[15] Free Software Foundation Inc., "GDB: The GNU Project Debugger," vol. 2006, 2006, http://www.gnu.org/software/gdb/.

[16] Y. Freund and R. E. Schapire, "A Short Introduction to Boosting," *Journal of Japanese Society for Artificial Intelligence*, vol. 14, no. 5, pp. 771-780, 1999.

[17] H. Gall, M. Jazayeri, and J. Krajewski, "CVS Release History Data for Detecting Logical Couplings," Proc. of 6th International Workshop on Principles of Software Evolution (IWPSE'03), Helsinki, Finland, pp. 13-23, 2003.

[18] S. Garfinkel, "History's Worst Software Bugs," 2005, http://wired.com/news/technology/bugs/0,2924,69355,00.html.

[19] M. W. Godfrey and L. Zou, "Using Origin Analysis to Detect Merging and Splitting of Source Code Entities," *IEEE Trans. on Software Engineering*, vol. 31, no. 2, pp. 166-181, 2005.

[20] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History," *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653-661, 2000.

[21] I. Guyon and A. Elisseeff, "An Introduction to Variable and Feature Selection," *Journal of Machine Learning Research*, vol. 3, pp. 1157-1182, 2003.

[22] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction," *IEEE Transactions on Software Engneering*, vol. 31, no. 10, pp. 897-910, 2005.

[23] A. E. Hassan and R. C. Holt, "The Top Ten List: Dynamic Fault Prediction," Proc. of 21st International Conference on Software Maintenance (ICSM 2005), Budapest, Hungary, pp. 263-272, 2005.

[24] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," Proc. of the 19th Object Oriented Programming Systems Languages and Applications (OOPSLA '04), Vancouver, British Columbia, Canada, pp. 92-106, 2004.

[25] J. S. Huggins, "First Computer Bug," 2006, http://www.jamesshuggins.com/h/tek1/first_computer_bug.htm.

[26] T. Joachims, "Text Categorization with Support Vector Machines: Learning with Many Relevant Features," Proc. of ECML 98, 10th European Conference on Machine Learning, Chemnitz, Germany, pp. 137-142, 1998.

[27] T. M. Khoshgoftaar and E. B. Allen, "Ordering Fault-Prone Software Modules," *Software Quality Control Journal*, vol. 11, no. 1, pp. 19-37, 2003.

[28] T. M. Khoshgoftaar and E. B. Allen, "Predicting the Order of Fault-Prone Modules in Legacy Software," Proc. of The Ninth International Symposium on Software Reliability Engineering, Paderborn, Germany, pp. 344-353, 1998.

[29] S. Kim, K. Pan, and E. J. Whitehead, Jr., "Micro Pattern Evolution," Proc. of Int'l Workshop on Mining Software Repositories (MSR 2006), Shanghai, China, pp. 40-46, 2006.

[30] S. Kim, K. Pan, and E. J. Whitehead, Jr., "When Functions Change Their Names: Automatic Detection of Origin Relationships," Proc. of 12th Working Conference on Reverse Engineering (WCRE 2005), Pennsylvania, USA, pp. 143-152, 2005.

[31] S. Kim, E. J. Whitehead, Jr., and J. Bevan, "Properties of Signature Change Patterns," Proc. of 22nd International Conference on Software Maintenance (ICSM 2006), Philadelphia, Pennsylvania, 2006.

[32] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead, Jr., "Automatic Identification of Bug Introducing Changes," Proc. of 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), Tokyo, Japan, 2006.

[33] R. Krovetz, S. Ugurel, and C. L. Giles, "Classification of Source Code Archives," Proc. of the 26th International ACM SIGIR Conference on Research and Development in Information Retrieval, Toronto, Canada, pp. 425-426, 2003.

[34] A. Kuhn, S. Ducasse, and T. Girba, "Enriching Reverse Engineering with Semantic Clustering," Proc. of 12th Working Conference on Reverse Engineering (WCRE 2005), Pittsburgh, Pennsylvania, USA, pp. 133-142, 2005.

[35] R. Kumar, S. Rai, and J. L. Trahan, "Neural-Network Techniques for Software-quality Evaluation," Proc. of 1998 Reliability and Maintainability Symposium, Anaheim, CA, pp. 155-161, 1998.

[36] T. K. Landauer, P. W. Foltz, and D. Laham, "Introduction to Latent Semantic Analysis," *Discourse Processes*, vol. 25, pp. 259-284, 1998.

[37] B. Livshits and T. Zimmermann, "DynaMine: Finding Common Error Patterns by Mining Software Revision Histories," Proc. of the 2005 European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, pp. 296-305, 2005.

[38] J. Lyle and M. Weiser, "Automatic Program Bug Location by Program Slicing," Proc. of 2nd International Conference on Computers and Applications, pp. 877-883, 1987.

[39] J. I. Maletic and N. Valluri, "Automatic Software Clustering via Latent Semantic Analysis," Proc. of 14th IEEE International Conference on Automated Software Engineering (ASE'99), Cocoa Beach, Florida, USA, pp. 251, 1999.

[40] A. Marcus and J. I. Maletic, "Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing," Proc. of the 25th International Conference on Software Engineering, Portland, Oregon, pp. 125-135, 2003.

[41] A. Miller, *Subset Selection in Regression*: Chapman & Hall /CRC, 2002.

[42] A. Mockus and L. G. Votta, "Identifying Reasons for Software Changes Using Historic Databases," Proc. of 16th International Conference on Software Maintenance (ICSM 2000), San Jose, California, USA, pp. 120-130, 2000.

[43] A. Mockus and D. M. Weiss, "Predicting Risk of Software Changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169-180, 2002.

[44] D. C. Montgomery, G. C. Runger, and N. F. Hubele, *Engineering Statistics*: Wiley, 2001.

[45] A. W. Moore, "Cross-Validation," 2005, http://www.autonlab.org/tutorials/overfit.html.

[46] N. Nagappan and T. Ball, "Use of Relative Code Churn Measures to Predict System Defect Density," Proc. of 27th International Conference on Software Engineering (ICSE 2005), Saint Louis, Missouri, USA, pp. 284-292, 2005.

[47] N. Nagappan, T. Ball, and A. Zeller, "Mining Metrics to Predict Component Failures," Proc. of 28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, pp. 452-461, 2006.

[48] Netcraft LTD, "Netcraft: Web Server Survey Archives," 2006, http://news.netcraft.com/archives/web_server_survey.html.

[49] T. J. Ostrand and E. J. Weyuker, "The Distribution of Faults in a Large Industrial Software System," Proc. of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, Roma, Italy, pp. 55-64, 2002.

[50] 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*, vol. 31, no. 4, pp. 340-355, 2005.

[51] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the Bugs Are," Proc. of 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, Boston, Massachusetts, USA, pp. 86-96, 2004.

[52] J. Platt, *In Advances in Kernel Methods - Support Vector Learning*: MIT Press, 1998.

[53] B. Raskutti, H. L. Ferrá, and A. Kowalczyk, "Second Order Features for Maximizing Text Classification Performance," Proc. of 12th European Conference on Machine Learning, Freiburg, Germany, pp. 419-430, 2001.

[54] I. Rish, "An Empirical Study of the Naive Bayes Classifier," Proc. of IJCAI-01 workshop on Empirical Methods in AI, Seattle, Washington, pp. 41-46, 2001.

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

[56] S. Scott and S. Matwin, "Feature Engineering for Text Classification," Proc. of the Sixteenth International Conference on Machine Learning, Bled, Slovenia, pp. 379-388, 1999.

[57] F. Sebastiani, "Machine Learning in Automated Text Categorization," *ACM Computing Surveys*, vol. 34, no. 1, pp. 1-47, 2002.

[58] J. Śliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes?" Proc. of Int'l Workshop on Mining Software Repositories (MSR 2005), Saint Louis, Missouri, USA, pp. 24-28, 2005.

[59] Q. Song, M. Shepperd, M. Cartwright, and C. Mair, "Software Defect Association Mining and Defect Correction Effort Prediction," *IEEE Trans. Software Engineering*, vol. 32, no. 2, pp. 69-82, 2006.

[60] J. C. Spohrer, E. Soloway, and E. Pope, "Where the Bugs Are," Proc. of the SIGCHI Conference on Human Factors in Computing Systems, San Francisco, California, USA, pp. 47-53, 1985.

[61] The Mozilla Organization, "Bugzilla," 2006, http://www.bugzilla.org/.

[62] V. N. Vapnik, *The Nature of Statistical Learning Theory*: Springer-Verlag, 1995.

[63] Wikipedia, "CamelCase," 2005, http://en.wikipedia.org/wiki/CamelCase.

[64] C. C. Williams and J. K. Hollingsworth, "Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques," *IEEE Trans. Software Engineering*, vol. 31, no. 6, pp. 466-480, 2005.

[65] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques (Second Edition)*: Morgan Kaufmann, 2005.

[66] A. Zeller, *Why Programs Fail*: Elsevier, 2006.

[67] L. Zhang, J. Zhu, and T. Yao, "An Evaluation of Statistical Spam Filtering Techniques," *ACM Transactions on Asian Language Information Processing (TALIP)*, vol. 3, no. 4, pp. 243-269, 2004.

[68] T. Zimmermann, S. Kim, A. Zeller, and E. J. Whitehead, Jr., "Mining Version Archives for Co-changed Lines," Proc. of Int'l Workshop on Mining Software Repositories (MSR 2006), Shanghai, China, pp. 72-75, 2006.

[69] T. Zimmermann and P. Weißgerber, "Preprocessing CVS Data for Fine-Grained Analysis," Proc. of Int'l Workshop on Mining Software Repositories (MSR 2004), Edinburgh, Scotland, pp. 2-6, 2004.

[70] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," *IEEE Trans. Software Engineering*, vol. 31, no. 6, pp. 429-445, 2005.