2011

Fuzzy set and cache-based approach for bug triaging

Ahmed Tamrawi
Iowa State University

Fuzzy set and cache-based approach for bug triaging

by

Ahmed Y. Tamrawi

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Tien N. Nguyen, Major Professor
Jennifer Davidson
Morris Chang

Iowa State University

Ames, Iowa

2011

DEDICATION

To the four pillars of my life: my parents, my wife, and my sisters. Without you, my life would fall apart. I might not know where the life's road will take me, but walking with You, through this journey has given me strength.

Mom, you have given me so much, thanks for your faith in me, and for teaching me that I should never surrender.

Daddy, you always told me to reach for the stars. I think I got my first one. Thanks for inspiring my love for computer.

Salwa, you are everything to me, without your love and understanding I would not be able to make it.

Alaa, Hanaa, and Aseel, you are the stars shining my sky and lightening my way to success and without you I would have never made it this far in life.

May Allah keep you all safe and happy.

We made it...

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my sincere thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Tien N. Nguyen for his guidance, patience and support throughout this research and the writing of this thesis. His insights and words of encouragement have often inspired me and renewed my hopes for completing my graduate education.

Second, Tung T. Nguyen for his great help throughout this research and the writing of this thesis.

I would also like to thank my committee members for their efforts and contributions: Dr. Morris Chang, and Dr. Jennifer Davidson.

Special thanks for my research group Jafar, Hoan, Tuan Anh, and Hung for their support and help throughout my research.

Finally, I would like to thank all my family members and friends for their love and support

ABSTRACT

Software bugs are inevitable and bug fixing is an essential and costly phase during software development. Such defects are often reported in bug reports which are stored in an issue tracking system, or bug repository. Such reports need to be assigned to the most appropriate developers who will eventually fix the issue/bug reported. This process is often called *Bug Triaging.*

Manual bug triaging is a difficult, expensive, and lengthy process, since it needs the *bug triager* to manually read, analyze, and assign bug fixers for each newly reported bug. Triagers can become overwhelmed by the number of reports added to the repository. Time and efforts spent into triaging typically diverts valuable resources away from the improvement of the product to the managing of the development process.

To assist triagers and improve the bug triaging efficiency and reduce its cost, this thesis proposes Bugzie, a novel approach for automatic bug triaging based on fuzzy set and cache-based modeling of the bug-fixing capability of developers. Our evaluation results on seven large-scale subject systems show that Bugzie achieves significantly higher levels of efficiency and correctness than existing state-of-the-art approaches. In these subject projects, Bugzie's accuracy for top-1 and top-5 recommendations is higher than those of the second best approach from 4-15% and 6-31%, respectively as Bugzie's top-1 and top-5 recommendation accuracy is generally in the range of 31-51% and 70-83%, respectively. Importantly, existing approaches take from hours to days (even almost a month) to finish training as well as predicting, while in Bugzie, training time is from tens of minutes to an hour.

CHAPTER 1 Introduction

A key collaborative hub for many software projects is a database of reports describing both bugs that need to be fixed and new features to be added[11]. This database is often called a bug repository [39] or issue tracking system. Such repositories determine which developer has expertise in different areas of the product, and it can help improve the quality of the software produced.

However, the use of a bug repository also has a cost. Developers can become overwhelmed with the number of reports submitted to the bug repository as each report needs to be assigned to the most appropriate developer who will be able to fix it. This process is known as *bug triaging* [2]. Each bug report is triaged to determine if it describes a valid problem and if so, the asignee of the bug needs to handle this bug into the development process by fixing the reported issues.

Manual bug triaging is a difficult, expensive, and lengthy process, since it needs the person who triages the reports - the *bug triager* - to manually read, analyze, and assign bug fixers for each newly reported bug. To assist triagers and support developers with the development-oriented decision they make during triage activities, this thesis proposes Bugzie, a novel fuzzy set and cache-based approach for automatic bug triaging.

The rest of this chapter proceeds as follows. First, we provide a brief overview of bug reports, followed by a brief overview of Bugzie, our automatic bug triaging approach. We conclude by outlining the contributions of this work.

1.1 An Overview of Bug Reports

A bug report contains a variety of information. Some of the information is categorical such as the report's identification number, its resolution status (i.e., new, unconfirmed, resolved), the product component the report is believed to involve and which developer has been given responsibility for the report. Other information is descriptive, such as the title of the report, the description of the report and additional comments, such as discussions about possible approaches to resolving the report. Finally, the report may have other information, such as attachments or a list of reports that need to be addressed before this report can be resolved.

1.2 Bugzie Overview

Bugzie considers a software system to have a collection of technical aspects/concerns, which are described via the corresponding technical terms appearing in software artifacts. Among the artifacts, a bug report describes an issue(s) related to some technical aspects/concerns via the corresponding technical terms. Thus, a potential/capable/relevant fixer for that report is the one that has bug-fixing capability/expertise/knowledge on the reported aspects. Therefore, in Bugzie, the key research question is that:

Given a bug report, how to determine who have the most bug-fixing capability/expertise with respect to the reported technical aspect(s).

The key idea of Bugzie is to model the *fixing correlation/association* of developers toward a technical aspect via fuzzy sets [24]. The fixing correlation/association represents the bug-fixing capability/expertise of developers with respect to the technical aspects in a project, in which the fuzzy sets are defined for the corresponding technical terms and built based on developers' past fixing bug reports and activities. Then, Bugzie recommends the most potential fixer(s) for a new bug report based on such information.

For a specific technical term t, a *fuzzy set* C_t is defined to represent the set of developers who have the bug-fixing expertise relevant to t, i.e. the most capable/competent ones to fix the bugs on the technical aspects described via the term t. The membership score of a developer d to C_t, i.e. the degree of certainty that d is a capable fixer for the bugs on the technical aspect(s) corresponding to t, is calculated via the similarity of the set of fixed bug reports containing t, and the set of bug reports that d has fixed. That is, the more distinct and prevalent the term t in the bug reports d has fixed, the higher the degree of certainty that d is a competent fixer for the technical issues corresponding to t. Then, for a new bug report B, the fuzzy set C_B of capable developers toward technical aspects reported in B is modeled by the union set of all fuzzy sets (over developers) corresponding to all terms extracted from B.

To cope with the large numbers of active developers and technical terms in large and long-lived projects, Bugzie has two design strategies on selecting the suitable *fixer candidates* and *significant terms* for the computation. Conducting an empirical study on several bug databases of real-world projects, we discovered the locality of the fixing activity: ”the recent fixing developers are likely to fix bug reports in the near future”. For example in Eclipse, 81% of actual fixers belong to the 10% developers having the most recent fixing activities. Thus, we propose to select a portion of recent fixers as the candidates for fixing a new bug report. In addition, instead of using all extracted words as terms for the computation, Bugzie is flexible to use only the terms that are highly correlated with each developer as the most significant terms to represent her/his fixing expertise.

To adapt with software evolution, Bugzie updates its model regularly (e.g. the lists of fixer candidates and terms, and the membership scores) as new information is available. We will discuss our approach and algorithms in details in chapters 3 and 4.

1.3 Thesis Contribution

This Thesis provides the following key contributions:

1. A scalable, fuzzy set and cache-based automatic bug triaging approach, which is significantly more efficient and accurate than existing state-of-the-art approaches;

2. The finding of the *locality* of fixing activity: one of the recent fixers is likely to be the fixer of the next bug report;

3. A comprehensive evaluation on the efficiency and correctness of Bugzie in comparison with existing approaches;

4. An observation/method to capture a small and significant set of terms describing developers' bug-fixing expertise.

5. A benchmark (bug datasets) and a tool-set for potential reproduced and enhanced approaches.

1.4 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2, we introduce the process of collecting our dataset and present an empirical study and a motivating example for our approach. Chapter 3 describes our approach for bug triaging in details. Chapter 4 describes the algorithms used in our approach. Chapter 5 presents our empirical evaluation and the comparison of our results to the state-of-the-art approaches. Chapter 6 discusses some related work and concludes the thesis.

CHAPTER 2 Empirical Study and Motivation

In this chapter, we will describe our data collection process for the study (section 2.1). Then, a case study example is presented to motivate our philosophy on the correlation between the fixing developers and the technical aspects reported in the bug reports (sections 2.2 and 2.3). Finally, we will present in detail our empirical study in which we found an important characteristic on the locality of bug fixing activities of developers (section 2.4). We will utilize these findings in the development of our approach and use the collected datasets for the evaluation.

2.1 Data Collection

Our datasets contain bug reports, corresponding fixers, and related information (e.g. summary, description, and creation/fixing time). Table 2.1 shows our collected datasets of seven projects: FireFox[14], Eclipse[13], Apache[3], Netbeans[30], FreeDesktop[16], Gcc[17], and Jazz. All bug reports and their data are available and downloaded from the bug tracking systems of the corresponding projects, except that Jazz data is available for us as a grant from IBM Corporation. We collected bug records noted as *fixed* and *closed*. Duplicated and unresolved (open) bug reports were excluded. Re-opened/un-finished bug fixes were not included either.

In Table 2.1, Column Time shows the time period of the fixed bug reports. Columns Report and Fixer show the number of fixed bug reports and that of the corresponding fixing developers, respectively. Two very large datasets (Eclipse and FireFox) have nearly two hundreds of thousands reports and thousands of bug fixers. The other datasets have between 20-50K records and 150-1,700 fixers.

Project	Time	Report	Fixer	Term
Firefox	04/07/1998 - 10/28/2010	188,139	3,014	177,028
Eclipse	10/10/2001 - 10/28/2010	177,637	2,144	193,862
Apache	05/10/2002 - 01/01/2011	43,162	1,695	110,231
NetBeans	01/01/2008 - 11/01/2010	23,522	380	42,797
FreeDesktop	01/09/2003 - 12/05/2010	17,084	374	61,773
Gcc	08/03/1999 - 10/28/2010	19,430	293	63,013
Jazz	06/01/2005 - 06/01/2008	34,228	156	39,771

Table 2.1: Statistics of All Bug Report Data

2.1.1 Bug Reports Pre-Processing

For each bug report, we extracted its unique bug ID, the actual fixing developer's ID, email address, creation and fixing time, summary, and full description. Comments and discussions are excluded. We merged the summary and description of each bug report. Then, using WVTool[40], we extracted their terms and preprocessed them, such as stemming for term normalization and removing grammatical and stop words. Column Term in Table 2.1 shows the total numbers of terms in all datasets.

2.2 A Motivating Example

Let us present a motivating example in our collected bug reports that leads to our approach for automatic bug triaging. Figure 2.1 depicts a bug report from Eclipse dataset, with the relevant fields including 1) a unique identification number of the report (ID), the fixing date (FixingDate), the fixing developer (AssignedTo), a short summary (Summary), and a full description (Description) of the bug.

ID:006021
FixingDate:2002-05-08 14:50:55 EDT
AssignedTo:James Moody
Summary:New Repository wizard follows implementation model, not user model.
Description:The new CVS Repository Connection wizard's layout is confusing. This is because it follows the implementation model of the order of fields in the full CVS location path rather than the user model...

Figure 2.1: Bug report #6021 in Eclipse project

The bug report describes an issue that the layout of the wizard for CVS repository connection was not properly implemented. Analyzing Eclipse's documentation, we found that this issue is related to a technical aspect: *version control and management* (VCM) for software artifacts. This aspect of VCM can be recognized in the report's contents via its descriptive terms such as CVS, repository, connection, and path. It is project-specific since not all systems have it. Checking the corresponding fixed code in Eclipse, we found that the bug occurred in the code implementing an operation of VCM: CVS repository connection. The bug was assigned to and fixed by a developer named *James Moody*.

Searching and analyzing other Eclipse' bug reports, we found that *James Moody* also fixed several other VCM-related bugs, for example, bug #0002 (Figure 2.2). The description states that the system always used its default editor to open any resource file (e.g. a GIF file) regardless of its file type. This aspect of VCM is described via the terms such as repository, resource, and editor. This observation suggests that *James Moody* probably has the expertise, knowledge, or capability with respect to fixing the VCM-related bugs in Eclipse.

ID:000002
FixingDate:2002-04-30 16:30:46 EDT
AssignedTo:James Moody
Summary:Opening repository resources doesn't honor type.
Description:Opening repository resource always open the default text editor and doesn't honor any mapping between resource types and editors. As a result it is not possible to view the contents of an image (*.gif file) in a sensible way....

Figure 2.2: Bug report #0002 in Eclipse project

2.3 Implications and our Approach

The example in previous section suggests us the following:

1. A software system has several technical aspects. Each could be associated with some descriptive technical terms. A bug report is related to one or multiple technical aspects.

2. If a developer frequently fixes the bugs related to a technical aspect, we could consider her/him to have *bug-fixing expertise/capability* on that aspect, i.e., (s)he could be a

capable/competent fixer for a future bug related to that aspect.

Based on those two implications, we approach to solve the problem of automated bug triaging using the following *key philosophy*:

"Who have the most bug-fixing capability/expertise with respect to the reported technical aspect(s) in a given bug report should be the fixer(s)".

Since technical aspects could be described via the corresponding technical terms, our solution could rely on the modeling of the fixing capability of a developer toward a technical aspect via the *association/correlation of that developer with the technical terms for that aspect.* Specifically, we will determine the most capable developers toward a technical aspect in the project based on their past fixing activities. Then, when a new bug is reported, we will recommend those developers who are most capable of fixing the corresponding technical issue(s) in the given bug report.

Our philosophy is different from existing approaches to automatic bug triaging [2, 7, 10, 28]. The philosophy from existing machine learning (ML)-based approaches [2, 7] is that if a new bug report is closest in characteristics/similarity with a set of bug reports fixed by a developer, (s)he should be suggested. That is, they characterize the classes of bug reports that each developer has fixed, and then classify a new bug report based on that classification. Another philosophy is from existing ML and information retrieval (IR) approaches [28, 10], which aim to profile a developer's expertise by a set of characteristic features (e.g. terms) in her/his fixed bug reports, and then match a new bug report with such profiles to find the fixer(s).

Our approach is centered around the association/correlation between two sets, *developers and terms*. Thus, in order to determine the most capable fixers with respect to the technical aspect(s) in a bug report, we have to address the questions of how to make the selections and take into account relevant terms and developers.

As shown in Table 2.1, for large projects with long histories, the numbers of terms (after stemming and filtering) are still very large (e.g. 200K words for FireFox). More importantly, not all terms appearing in a bug report would be technically meaningful and relevant to the fixers or reported technical issues. Thus, using all of them would be computationally expensive, and even worse, might reduce the fixer recommendation accuracy by introducing noise to the ranking. The motivating example suggests that such term selection could be based on the level of association, i.e. a term having high correlation with some developers could be a significant term for bug triaging, e.g. the association of repository and *James Moody* (Details will be presented in Chapter 3).

The selection of developers is also needed because in a large and long-lived project, the number of developers could be large and some might not be as active in certain technical areas as others any more. Moreover, considering all developers as the fixer candidates for a bug report could be computationally costly.

Next, we will describe an empirical study that motivates our developers' selection strategy.

2.4 Locality of Fixing Activity

Analyzing several bug reports fixed by the same person in our datasets, we found that (s)he tends to have recent fixing activities. For example in Eclipse dataset, bug reports #312322, #312291, #312466, and #311848 were fixed by the same fixer *Darin Wright* in two days 05/10 and 05/11/2010. We hypothesize that:

The fixing activity has locality, i.e. a developer having recent fixing activities has higher tendency to fix some newly bug reports than developers with less recent fixing ones (*the recent fixing developers are likely to fix bug reports in the near future*).

To validate this hypothesis, we have conducted an experiment in which we analyzed the collected datasets to compute how often a fixer of a bug report is the one who has some recent fixing activity. First, we chronically sorted the bug reports in a project by their fixing time. For a bug report b that was fixed at time t by a developer d, we sorted all developers having fixing activities before t based on their most recent fixing time, i.e. a developer performing a fix more recently to time t was sorted higher. Then, if d belongs to the top $x\%$ fixers of that list, we count this as a *hit*. Finally, we compute $p(x)$ as the percentage of hits over the total number of analyzed bug reports.

Table 2.2 shows the experiment result for all projects. As seen, it is consistent in all systems

Recent	**Eclipse**	**Firefox**	**Jazz**	**Gcc**	**Apache**	**FreeDesktop**	**NetBeans**
10%	81%	82%	62%	84%	71%	73%	69%
20%	87%	92%	74%	92%	81%	89%	87%
30%	92%	96%	83%	95%	89%	94%	94%
40%	96%	97%	92%	97%	92%	96%	96%
50%	98%	98%	97%	98%	94%	97%	97%
60%	98%	98%	99%	98%	95%	98%	98%
70%	99%	98%	99%	98%	96%	98%	98%
80%	99%	98%	100%	99%	96%	98%	98%
90%	99%	98%	100%	99%	96%	98%	98%
100%	99%	98%	100%	99%	96%	98%	99%

Table 2.2: Percentage of Actual Fixers having Recent Fixing Activities

that $p(x)$ is rather large even at small x. For example in Eclipse, at $x = 10\%$, $p(x) = 81\%$, i.e. in around 81% of the cases, the fixer of a bug report is in the top 10% of the developers who have most recent fixing activities. At $x = 50\%$, $p(x)$ exceeds 97% in 6 systems. Note that, at $x = 100\%$, $p(x)$ could not reach 100% since there are always new fixers who have no historical fixing activity, thus, (s)he does not belong to the list of developers with recent fixing activities.

2.4.1 Implications

The experiment result confirms our hypothesis on the locality of fixing activity. This result suggests that: instead of selecting all available developers as fixer candidates for a bug report, we could select a small portion of them based on their recent fixing activities. This selection would significantly improve time efficiency without losing much accuracy.

Next, we will discuss in detail Bugzie, our automatic bug triaging approach.

CHAPTER 3 Bugzie Model

3.1 Overview

In Bugzie, the problem of automatic bug triaging is modeled as follows:

Given a bug report, find the developer(s) with the most fixing capability/expertise with respect to the reported technical issue(s).

Existing approaches view this problem as a classification problem: each developer is considered as a class for bug reports in which their characteristics are learned via her/his past fixed reports. An unfixed bug report will be assigned to the developer(s) corresponding to the most relevant/similar class(es) to the report.

In contrast, Bugzie considers this as a ranking problem:

For each given bug report, Bugzie determines a ranked list of developers who are most capable of handling the reported technical issue(s).

Thus, instead of learning the characteristics of each class/developer based on her/his past fixed reports, Bugzie determines and ranks *the fixing capability/expertise* of the developers *toward the technical aspects* by modeling the *correlation/association of a developer and a technical aspect.* That is, if a developer has higher fixing correlation with a technical aspect, (s)he is considered to have higher capability/expertise on that aspect, and (s)he will be ranked higher.

Because ”technical aspect” is an abstract concept, with potential different levels of granularity, Bugzie models them via their corresponding descriptive technical terms. That is, a technical aspect is considered as *a collection of technical terms* that are extracted directly from the software artifacts in a project, and more specifically from its bug reports.

Bugzie utilizes the fuzzy set theory [24] to model the *fixing correlation/association between* *developers and the technical terms/aspects*, which is used to recommend the most capable fixers for a given bug report. Bugzie also uses the locality of fixing activity to select the fixer candidates, and uses the levels of correlation between the fixers and terms to identify the most correlated/important terms for each developer.

3.2 Association of Fixer and Term

Definition 1 (Capable Fixer toward A Term) *For a specific technical term t, a fuzzy set C_t, with associated membership function $\mu_t()$, represents the set of capable fixers toward t, i.e. developers who have the bug-fixing expertise relevant to technical aspect(s) described by t.*

In fuzzy set theory, fuzzy set C_t is determined via a membership function μ_t with the values in the range of [0,1]. For a developer d, the membership score $\mu_t(d)$ determines the certainty degree of the membership of d in C_t, i.e. how likely d belongs to the fuzzy set C_t. In this context, $\mu_t(d)$ determines the degree to which d is capable of fixing the bug(s) relevant to the technical aspect(s) associated with t. The membership score also determines the ranking, i.e. if $\mu_t(d) > \mu_t(d')$ then d is considered to be more capable than d' in the issues related to t. $\mu_t(d)$ is calculated based on d's past fixing activities as follows:

Definition 2 (Membership Score toward a Term) *The membership score $\mu_t(d)$ is calculated as the correlation between the set D_d of the bug reports d has fixed, and the set D_t of the bug reports containing term t:*

$$\mu_t(d) = \frac{|D_d \cap D_t|}{|D_d \cup D_t|} = \frac{n_{d,t}}{n_t + n_d - n_{d,t}}$$

In this formula, n_d, n_t, and $n_{d,t}$ are the number of bug reports that d has fixed, the number of reports containing the term t, and that with both, respectively (counted from the available training data, i.e. given fixed bug reports).

With this formula, the value of $\mu_t(d) \in [0, 1]$. The higher $\mu_t(d)$ is, the higher the degree that d is a capable fixer for the bugs related to term t. If $\mu_t(d) = 1$, then only d had fixed the bug reports containing t, thus, d is highly capable of fixing the bugs relevant to the technical aspects associated with term t. If $\mu_t(d) = 0$, d has never fixed any bug report containing t, thus, might not be the right fixer with respect to t. In general cases, the more frequently a term t appears in the reports that developer d has fixed, the higher $\mu_t(d)$ is, i.e. the more likely that developer d has fixing expertise toward the technical aspects associated to t.

The membership value $\mu_t(d)$, representing the *fixing correlation of a developer toward a technical term*, is an intrinsically gradual notion, rather than a concrete one as in conventional logic. That is, the boundary for the set of developers who are capable of fixing the bug(s) relevant to a term t is fuzzy.

The membership score formula in Definition 2 allows Bugzie to favor (rank higher) the developers who have emphasized fixing activities toward some technical aspect/term t (i.e. specialists) over the ones with less specialization with their fixing activities on multiple other technical issues (i.e. generalists). That is, if both d and d' have similar levels of fixing activities on t, i.e. $n_{d,t}$ and $n_{d',t}$ are similar, but d' fixes on several other technical issues while d mostly emphasizes on t, then $n_{d'}$ will be much larger than n_d, and $\mu_t(d')$ will be smaller than $\mu_t(d)$. Thus, Bugzie will favor the specialist d.

Because a bug report might contain multiple technical issues/aspects, and each technical aspect could be expressed via multiple technical terms, Bugzie needs to model the capable fixers with respect to a bug report based on their correlation values toward its associated terms. This is done using the union operation in fuzzy set theory as follows.

Definition 3 (Capable Fixer for a Bug Report) *For a given bug report B, fuzzy set C_B, with associated membership function $\mu_B()$, represents the set of capable fixers for B, i.e. the developers who have the bug-fixing expertise relevant to technical aspect(s) reported in B. C_B is computed as the union of the fuzzy sets for the terms extracted from B*

$$C_B = \bigcup_{t \in B} C_t$$

In fuzzy set theory, union is a flexible combination, i.e. the strong membership to some sub-fuzzy set(s) will imply the strong membership to the combined fuzzy set. Especially, the more the sub-fuzzy sets with strong membership degrees, the stronger the membership of the combined fuzzy set is. According to [24], the membership score of the union set C_B is calculated as the following:

Definition 4 (Membership Score for a Report) *The membership score $\mu_B(d)$ is computed as the combination of the membership scores $\mu_t(d)$ of its associated terms t:*

$$\mu_B(d) = 1 - \prod_{t \in B} (1 - \mu_t(d))$$

$\mu_B(d)$ represents the fixing correlation of d toward bug report B. As seen, $\mu_B(d)$ is also within [0,1] and represents the degree in which developer d belongs to C_B, i.e. the set of capable fixers of the bug(s) reported in B. The value $\mu_B(d) = 0$ when all $\mu_t(d) = 0$, i.e. d has never fixed any report containing any term in B. Thus, Bugzie considers that d might not be as suitable as others in fixing technical issues reported in B. Otherwise, if there is one term t with $\mu_t(d) = 1$, then $\mu_B(d) = 1$, and d is considered as the capable developer (since only d has fixed bug reports with term t before). In general cases, the more the terms in B with high $\mu_t(d)$ scores, the higher $\mu_B(d)$ is, i.e. the more likely d is a capable fixer for bug report B. Using this formula, after calculating fixing correlation scores $\mu_B(d)$s for candidate developers, Bugzie ranks and recommends the top-scored developers as the most capable fixers for bug report B.

The union operation allows Bugzie to take into account the co-occurring/correlated terms associated with some technical aspects and reduce the impact of noises. Generally, a technical aspect could be expressed in some technical terms, such as the concern of version control in Eclipse might be associated with terms like $t =$ repository and $t' =$ cvs. Thus, these two terms tend to co-occur in the bug reports on version control and if a concrete bug report B contains both terms, B should be considered to be more relevant to version control than the ones containing only one term. That means, if d is a developer with fixing expertise in version control, $\mu_t(d)$ and $\mu_{t'}(d)$ should be equally high, and $\mu_B(d)$ must be higher than either of them. Those are actually true in our model. Since t and t' tend to co-occur, bug reports contain t, including the ones fixed by d, might also contain t'. Thus, two sets D_t and $D_{t'}$ are similar, and because d has fixing expertise on version control, $\mu_t(d)$ and $\mu_{t'}(d)$ will be similarly high. Assume that

$\mu_t(d) = 0.7$ and $\mu_{t'}(d) = 0.6$. Then, $\mu_B(d)$ = 1 - (1-0.7)*(1-0.6) = 0.88, i.e. higher than $\mu_t(d)$ and $\mu_{t'}(d)$.

Value $\mu_B(d)$ is not affected much by noises, i.e. the terms irrelevant to developers' expertise/technical aspects (e.g. misspelled words). Assume that B contains t and a noise e. Since e rarely occurs in the bug reports a developer d fixed, d has small membership score toward e, e.g. 0.1. Then, $\mu_B(d)$=1-(1-0.7)*(1-0.1)= 0.73, i.e. not much larger than $\mu_t(d)$= 0.7.

3.3 Fixer Candidate and Term Selection

In this section, we discuss our design strategies in Bugzie to select the suitably small sets of candidate fixers and significant/relevant terms to reduce the computation.

3.3.1 Selection of Fixer Candidates

The locality of fixing activity suggests:

The actual fixer for a given bug report is likely the one having recent fixing activity.

Thus, for each bug report, Bugzie chooses the top x% of developers sorted by their latest fixing time as the fixer candidates $F(x)$ for its computation. This is a trade-off between performance and accuracy. If x = 100%, all developers will be considered, accuracy could be higher, however, running time will be longer. Importantly, in general cases, the locality of fixing activity suggests that the loss in accuracy is acceptable. For example, from Table 2.2, by selecting x = 50%, we could reduce in half the computation time, while losing at most 1-3% of accuracy for all subject systems (by comparing the numbers in 50% and 100% lines).

3.3.2 Selection of Descriptive Terms

Following its fuzzy-based modeling, Bugzie measures the significance/descriptiveness based on the fixing correlation, i.e. the membership scores. That is, for a developer d and a term t, the higher their correlation score $\mu_t(d)$, the higher significance of t in describing the technical aspects that d has fixing capability/expertise. Thus, Bugzie selects the descriptive terms as follows. For each developer d, it sorts the terms in the descending order based on the correlation scores $\mu_t(d)$, and selects the top k terms in the sorted list as the significant terms $T_d(k)$ for developer d. The collection $T(k)$ of all such terms selected for all developers is considered as the set of technical terms for the whole system. Then, when recommending, Bugzie uses only those terms in its ranking formula. In other words, if a term extracted from the bug report under consideration does not belong to that list, Bugzie will discard it in the formulas in Section 3.2.

Table 3.1 shows such lists of top-10 terms having highest correlation scores with some Eclipse's developers produced by our tool. As seen, Bugzie discovers that *James Moody* has many fixing activities toward VCM technical aspect.

Ed Merks	**Darin Wright**	**Tod Creasey**	**James Moody**
xsd	debug	marker	outgoing
ecore	breakpoint	progress	vcm
xsdschema	launch	decoration	itpvcm
genmodel	console	dialog	repository
emf	vm	workbench	history
xsdecorebuild	memory	background	ccv
xmlschema	jdi	font	team
eobject	suspend	view	cvs
xmlhandler	config	ui	merge
ecoreutil	thread	jface	conflict

Table 3.1: Term Selection for Eclipse's developers

CHAPTER 4 Bugzie's Algorithms

This chapter describes the key algorithms in Bugzie. Given the model in Chapter 3 with two adjustable parameters x (for fixer candidates) and k (for selected term lists), Bugzie operates in three main phases: 1) **Initial Training**, i.e. building the fuzzy sets for the technical terms collected from the initially available information (e.g. already-fixed bug reports); 2) **recommending**, i.e. producing a ranked list of developers capable of fixing an unfixed bug report, and 3) **updating**, i.e. updating the fuzzy sets as new information is available (i.e. newly fixed bug reports).

4.1 Initial Training

In this phase, Bugzie uses a collection of already-fixed bug reports to build its initial internal data, including 1) the fuzzy sets of capable fixers for the available technical terms, 2) the fixer candidate list $F(x)$, 3) the individual term lists $T_d(k)$, and 4) the system-wide term list $T(k)$. While modeling the fuzzy sets, it stores only the counting values n_d, n_t, and $n_{d,t}$ (see Definition 2) for any available developer d and technical term t. The values $\mu_t(d)$ are computed on-demand to reduce the memory needed to store membership scores, and make the updating phase simpler (since only those counting numbers need to be updated).

4.2 Recommending

In this phase, Bugzie recommends the most capable developers for a given unfixed bug report B. First, it extracts all terms from B and keeps only terms belonging to the selected term list $T(k)$. Then, it computes the membership scores of all developers in the candidate list $F(x)$ using Definition 2. The values $\mu_t(d)$ are computed as needed using the counting values

n_d, n_t, and $n_{d,t}$. Finally, Bugzie ranks those membership scores and recommends the top-n developers as the most capable fixers for the bug(s) reported in B.

4.3 Updating

In this phase, Bugzie incrementally updates its internal data with newly available information (i.e. new bug reports are fixed by some developers). First, it updates the counting values n_d, n_t, and $n_{d,t}$ using newly available fixed bug reports by adding new corresponding counts for the new data. For example, if developer d just fixed a bug report B, Bugzie increases the counting number n_d by 1 and increases $n_{d,t}$, and n_t by 1 for any term t extracted from B. If a new term or a new developer just appears in new data, Bugzie creates new counting numbers n_t or n_d and $n_{d,t}$.

After updating the counting numbers, Bugzie updates the list $F(x)$, $T_d(k)$, and $T(k)$. Instead of re-sorting all available developers and terms to update those lists, Bugzie uses a caching strategy: it stores $F(x)$ as a cache (called *developer cache*). Thus, for each fixed bug report in the updating data, if the fixer does not belong to the cache, Bugzie will add it to the cache, and if the cache is full, it will remove from the cache the developer(s) having the *least* recent fixing activity.

Similarly, Bugzie also stores $T_d(k)$ as caches (called *term cache*), and updates them based on the membership scores. $T_d(k)$ is stored as a descendingly sorted list. During updating, if a term t does not belong to the cache and its score $\mu_t(d)$ is larger than that of some term currently in the cache, Bugzie will insert it to the cache, and if the cache is full, it will remove the least-scored term.

This updating and caching strategy makes our incremental updating very efficient. Importantly, it fits well with software evolution nature. The membership score $\mu_t(d)$ is computed on-demand with the most recently updated counting numbers n_d, n_t, and $n_{d,t}$. The cache $F(x)$ always reflects the developers having most tendency for fixing bugs. The lists $T_d(k)$ always consist of the terms having highest association with the developers. Existing approaches are not sufficiently flexible to support such caches of developers and terms. In Bugzie, during software evolution, time-sensitive knowledge on developers' fixing activities and important terms can be taken into account. In future work, other cache replacement strategies as in BugCache [23] could be explored.

Next, we will describe and discuss our empirical evaluation results on the collected datasets, and compare it with the state-of-the-art approaches.

CHAPTER 5 Empirical Evaluation

We evaluated Bugzie on our collected datasets (Section 2.1), some of which were used in prior bug triaging research [2, 28, 7]. We evaluated it with various parameters for developers' and terms' selections, and compared it with state-of-the-art approaches [12, 2, 28, 7]. All experiments were run on a Windows 7, Intel Core 2 Duo 2.10Ghz, 4GB RAM desktop.

5.1 Experiment Setup

To simulate the usage of Bugzie in practice, we used the same longitudinal data setup as in [7]. That is, all extracted bug reports from each bug repository in Table 2.1 were sorted in the chronological order of creation time, and then divided into 11 non-overlapped and equally sized frames.

Initially, frame 0 with its bug reports were used in initial training. Then, Bugzie used that training data to recommend a list of top-n developers to fix the first bug report in frame 1, $BR_{1,1}$. After that, we performed updating for our training data with tested bug report $BR_{1,1}$, and started recommending for the following bug report in frame 1, $BR_{1,2}$. After completing frame 1, the updated training data was then used to test frame 2 in the same manner. We repeated this until all the bug reports in all frames were consumed.

If a recommendation list for a bug report contains its actual fixer, we count this as a *hit* (i.e. a correct recommendation). For each frame under test, we calculated *prediction accuracy* as in [7]:

Definition 5 (Prediction Accuracy) *The ratio between the number of prediction hits over the total number of prediction cases.*

For example, if we have 100 bugs to recommend fixers for and for 20 of those bugs, we could recommend the actual fixing developer as the first developer in our recommendation list, the prediction accuracy for Top-1 is 20%; similarly, if the actual fixing developer is in our Top-2 for 60 bugs, the Top-2 prediction accuracy is 60%.

Then, We calculated the average accuracy value on all 10 frames for each choice of the top-ranked list of n. We also measured the training (initial training and updating) and recommending time.

5.2 Selection of Fixer Candidates

In this experiment, we tuned different options for the selection of fixer candidates (i.e. developer cache). Recall from Chapter 3 that Bugzie allows to choose x% of top fixers having most recent fixing activities. We ran it with various values of x%, increasing from 1-100% (at x=100%, all developers in the project's history were chosen). For each value of x, we measured prediction accuracy and total *processing time* (for training and recommending). The same process was applied for all datasets in Table 2.1.

Figures 5.1 and 5.2 show the graphs for the top-1 and top-5 prediction accuracy for different values of x for all datasets. As seen, all graphs exhibit the same behavior. The accuracy peaks at some value x that is quite smaller than 100%. In all 7 projects, accuracy reaches its peak at $x < 40\%$. This implies that selecting a suitable portion of recent fixers as candidates actually does not lessen much the accuracy. In some cases, it *improves* the prediction accuracy. For example, in FireFox, at $x = 20\%$, Bugzie has top-5 accuracy of 72.4%, while top-5 accuracy at $x = 100\%$ is only 70.7%, i.e. when considering all available fixers as candidates.

Definitely, selecting only a portion of available fixers as candidates also significantly improves time efficiency. Figure 5.3 displays the total processing time for all systems, which includes training and prediction time. Since in prediction/recommendation phase, Bugzie just needs to compute membership scores based on the stored counting values, prediction time is just a few tens of seconds for all cases. As seen, the processing time for FireFox and Eclipse is higher than that for other projects due to their large datasets. However, for FireFox, at

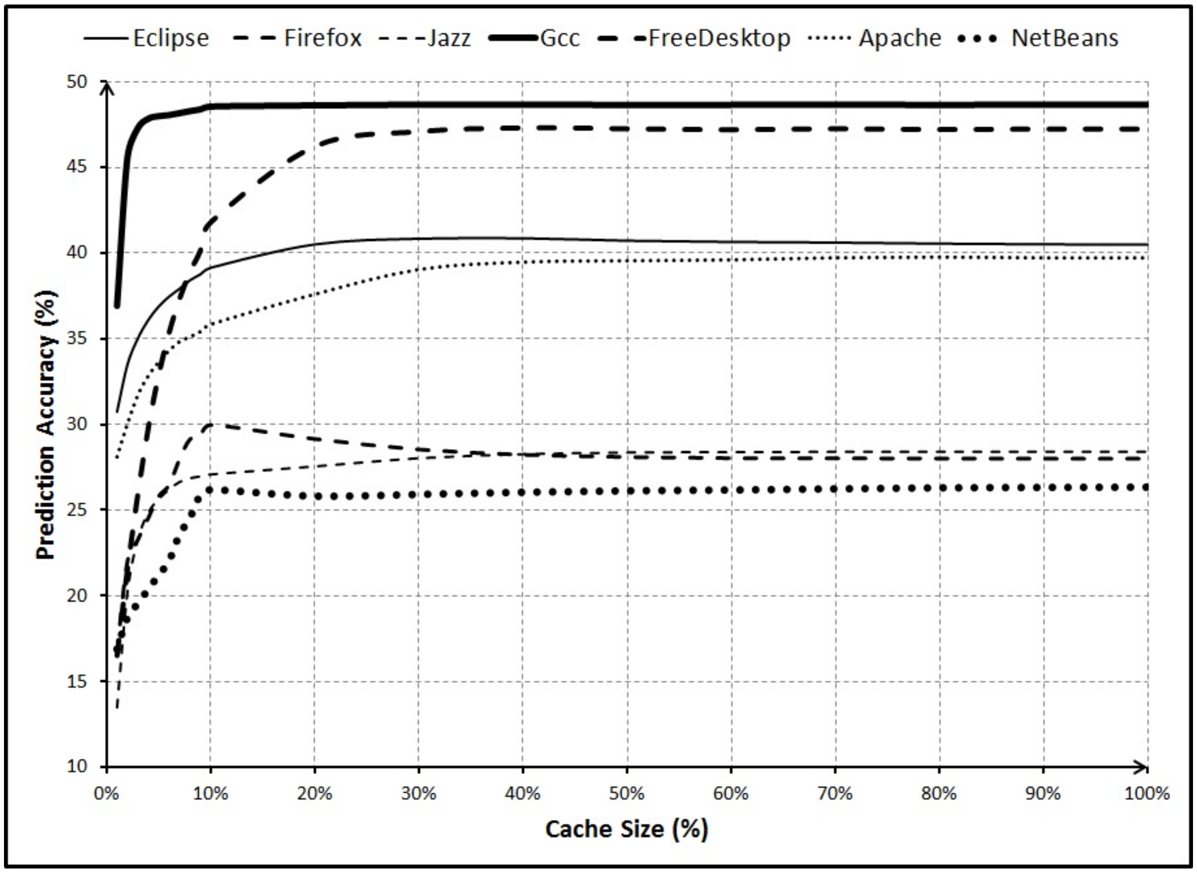

Figure 5.1: Top-1 Accuracy with Various Cache Sizes

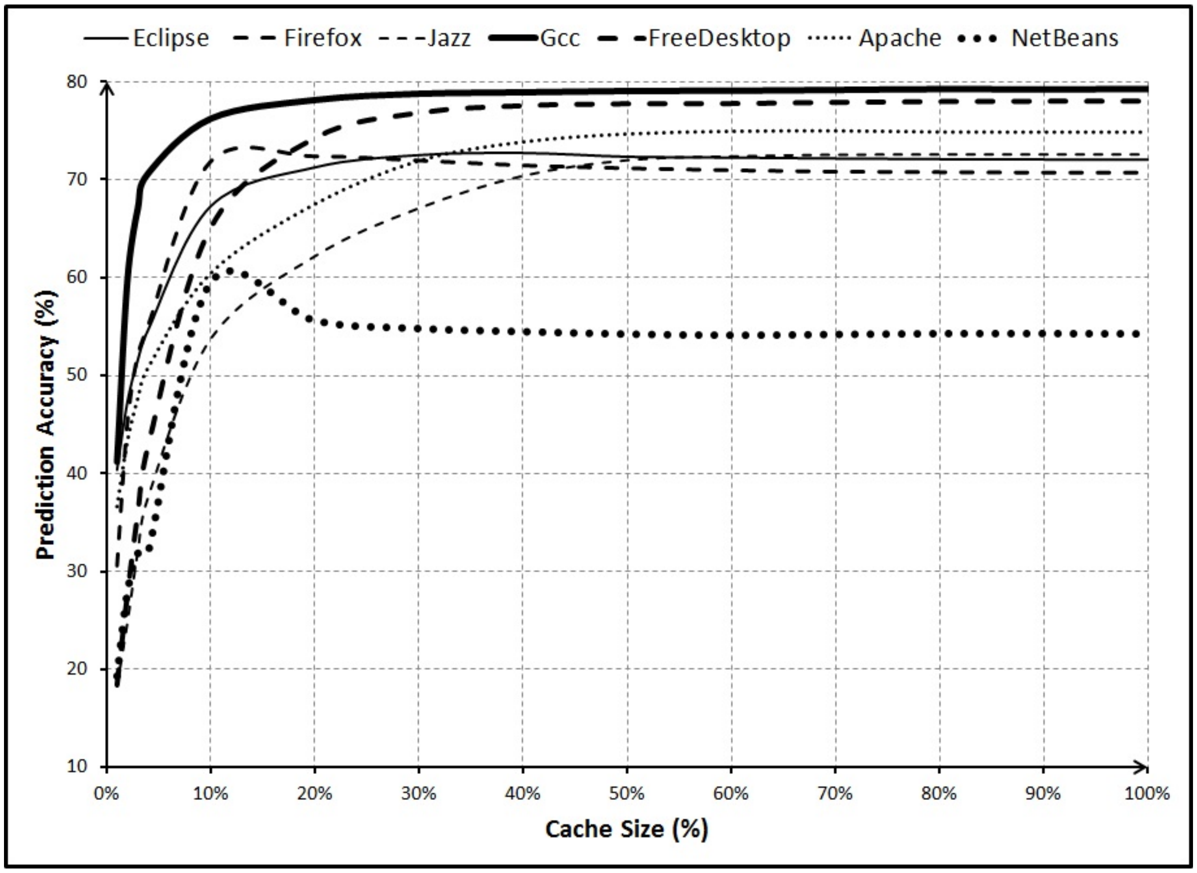

Figure 5.2: Top-5 Accuracy with Various Cache Sizes

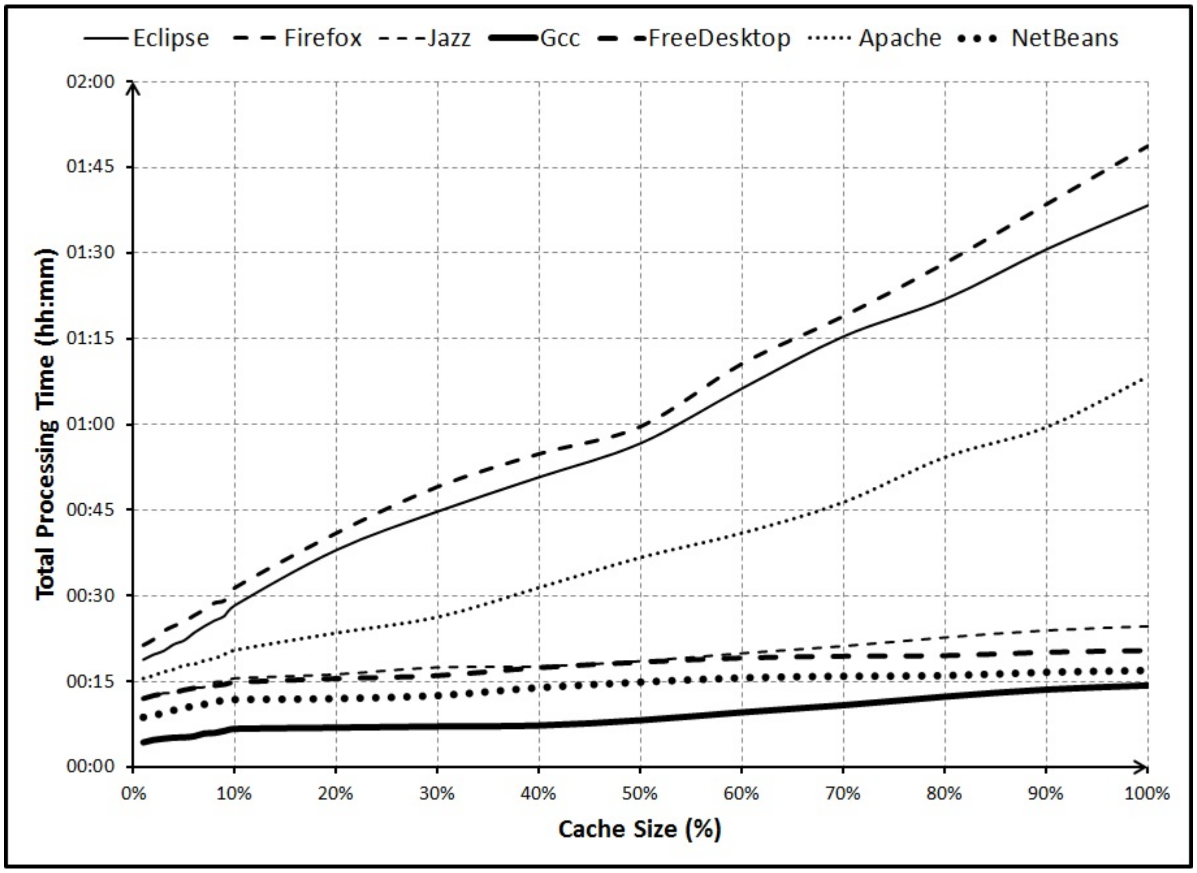

Figure 5.3: Processing Time with Various Cache Sizes

$x = 20\%$, with caching, Bugzie can reduce the processing time around 2.7 times less. The processing time is also linear with respect to the cache size of fixer candidates.

This result suggests that the selection of fixer candidates (i.e. developer cache) significantly improve time efficiency because Bugzie just needs to process a smaller number of developers. In some cases, it even helps improve prediction accuracy. We examined those cases and found that Bugzie fits well with the nature of the locality in fixing activity: the appropriate cache was able to capture the majority of actual fixers. Also, it did not include the developers who had high fixing expertise in some technical aspect in a very long time ago, but do not handle much that technical issue anymore. When including such developers and their past fixing terms, ranking could be imprecise since more irrelevant developers and terms are considered. As seen in Figure 5.2, the appropriate sizes of developer cache depend on individual projects.

5.3 Selection of Terms

We conducted a similar experiment for the selection of terms. Bugzie is flexible to allow the selection of only top-k terms that are most correlated with each fixer via their correlation/membership scores in the ranking process (Definition 2). We ran Bugzie with different values of k, increasing from 1-5,000. With k=5,000 for each developer, the system-wide term list $T(k)$ mostly covers all available terms in all bug reports. If a developer has the number of terms less than k, all of his associated terms with non-zero correlation scores are used. For each value of k, we measured top-n prediction accuracy and the total processing time. This procedure was applied for all systems in Table 2.1.

Figures 5.4 and 5.5 show the results of top-1 and top-5 prediction accuracy on all datasets, with different values of k. As seen, for all projects (except Apache), the graphs have similar shapes. This exhibits a very interesting phenomenon: accuracy increases and reaches its peak in the range of *3-20 terms*, and when more terms are used, accuracy slightly decreases to a stable level. Thus, selecting a small yet significant set of terms for ranking computation in fact *improves* prediction accuracy. For example, for Eclipse, at $k = 16$, we have top-5 accuracy of 80%, while at $k = 5,000$ (almost all extracted terms are included), top-5 accuracy is only 72%.

This result shows that the selection of terms could improve much prediction accuracy. The result also suggests that one just needs *a small yet significant set of terms for each developer* to describe his bug-fixing expertise. Bugzie with term selection is flexible to capture those significant terms representing the technical issues handled by each developer. For example, analyzing Eclipse's bug reports, we verified the core bug-fixing technical expertise of the fixers listed in Table 3.1. Bugzie also enables the exclusion of a large number of un-important terms in bug reports, as well as the terms with small correlation scores to developers. Those terms could have brought noises to the computation in Bugzie.

More importantly, selecting only a small portion of available terms also significantly improves time efficiency. Figure 5.6 shows the graph for the total processing time. As seen, in Eclipse, at $k = 16$ (the system-wide term list $T(k)$ has 6,772 terms), Bugzie is four times faster than at $k = 5,000$ ($T(k)$ has 193,862). Moreover, the processing time is also linear with respect

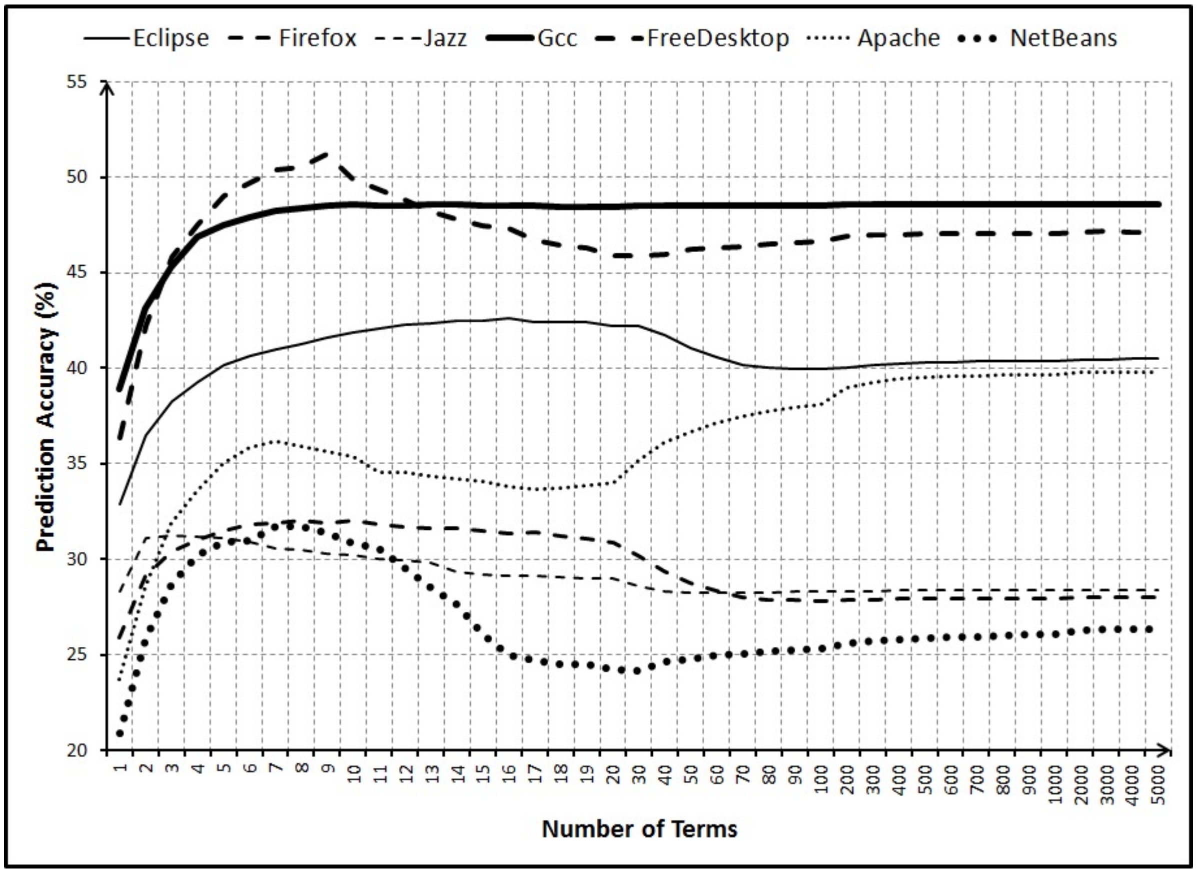

Figure 5.4: Top-1 Accuracy - Various Term Selection

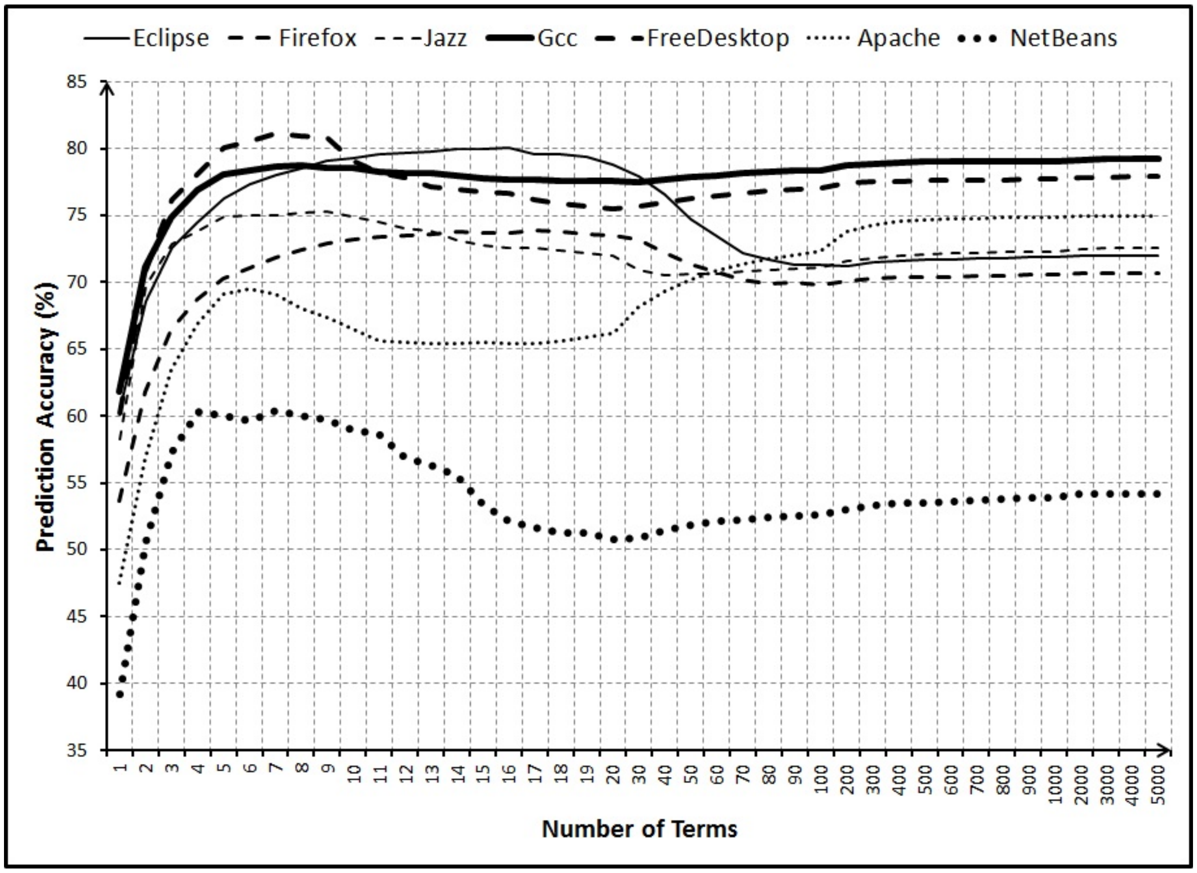

Figure 5.5: Top-5 Accuracy - Various Term Selection

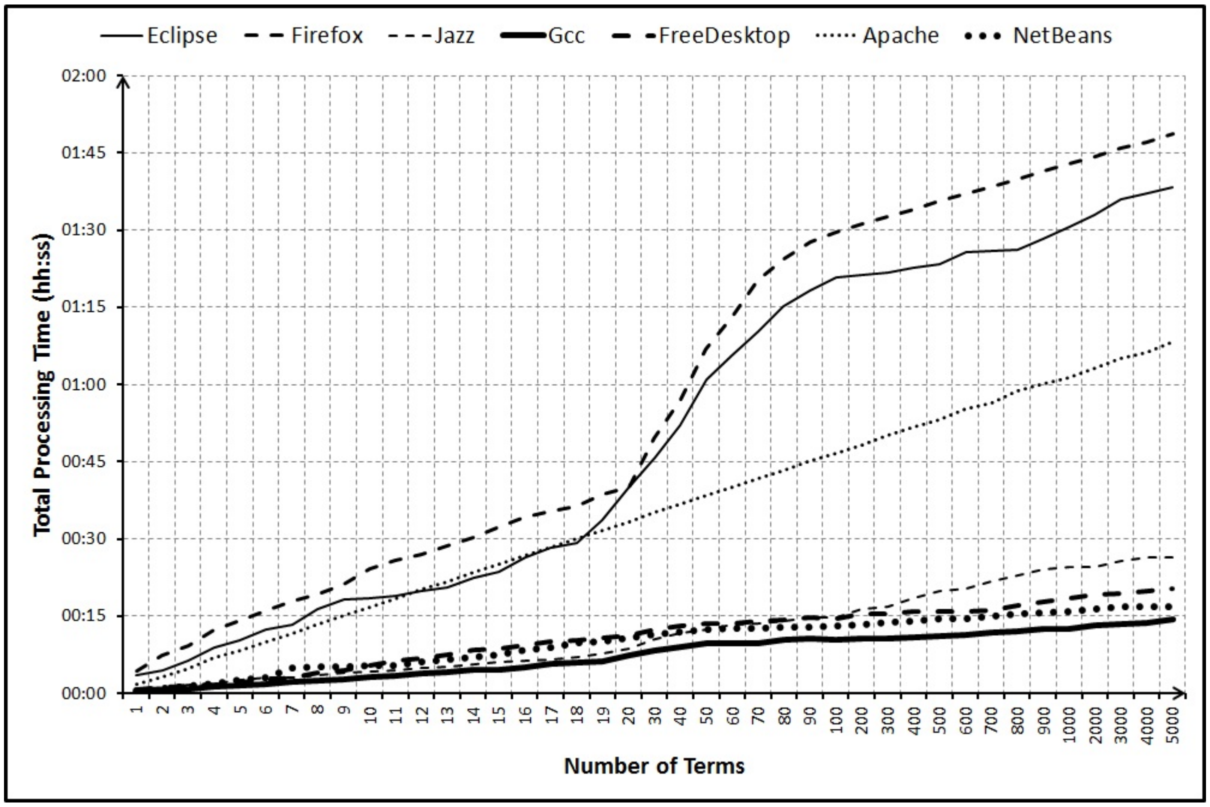

Figure 5.6: Processing Time - Various Term Selection

to the cache size of selected terms, showing that Bugzie is scalable well to large projects.

In Apache case, accuracy does not reach its highest point until $k = 300$. Examining the dataset, we found that Apache has a large number of developers (1,695), a medium number of bug reports (43,162), and a large number of terms (110,231). To correlate well a developer's expertise toward a bug report, Bugzie needs more terms than other subjects.

5.4 Selection of Developers and Terms

To evaluate the impacts of both types of selection (i.e, Candidates and Terms Selection), we conducted another experiment and tuned the model with different sizes of developer cache and term cache to get the better results. For each subject system in Table 2.1, we ran Bugzie on all datasets with all combinations of the best values we discovered in the previous experiments as the model's parameters/configurations. Tables 5.1, 5.2, and 5.3 show the accuracy and the total processing time with different parameters for 3 subject systems: Eclipse, Firefox, and FreeDesktop.

Tuning Parameters	Top -1	Top-2	Top-3	Top-4	Top-5	Time
$x = 40\%$, $k = 16$	45.0	61.2	71.2	78.2	83.2	12:00
$x = 100\%$, $k =$ All	40.5	53.7	61.7	67.5	72.0	1:39:12

Table 5.1: Eclipse: Accuracy - Various Parameters

Tuning Parameters	Top -1	Top-2	Top-3	Top-4	Top -5	Time
$x = 10\%$, $k = 10$	34.6	50.9	61.8	70.3	76.7	6:16
$x = 10\%$, $k = 17$	33.8	50.4	61.8	70.3	76.8	8:57
$x = 10\%$, $k = 18$	33.6	50.3	61.7	70.2	76.7	9:51
$x = 20\%$, $k = 10$	34.1	50.5	61.8	70.7	77.7	9:17
$x = 20\%$, $k = 17$	33.2	50.1	61.8	70.8	77.8	12:04
$x = 20\%$, $k = 18$	33.0	49.9	61.7	70.8	77.7	13:10
$x = 100\%$, $k =$ All	28.0	44.7	55.8	64.1	70.7	1:50:04

Table 5.2: FireFox: Accuracy - Various Parameters

Tuning Parameters	Top -1	Top-2	Top-3	Top-4	Top -5	Time
$x = 40\%$, $k = 7$	50.5	65.5	72.4	76.9	79.9	1:08
$x = 40\%$, $k = 9$	50.9	65.3	72.0	76.4	79.3	1:39
$x = 90\%$, $k = 7$	50.2	65.2	72.5	77.2	80.3	2:07
$x = 90\%$, $k = 9$	50.7	65.3	72.4	76.8	79.8	3:02
$x = 100\%$, $k =$ All	47.1	61.7	69.1	74.3	77.9	20:35

Table 5.3: FreeDesktop: Accuracy - Various Parameters

As seen, Bugzie could be tuned to achieve very high levels of accuracy and efficiency. For example, for Eclipse, the best configured model processes the whole Eclipse's bug dataset (with around 178K bug records and 2K developers) in only 12 minutes and achieve 83% top-5 prediction accuracy. That is about 9 times faster, and 11% more accurate than the base model ($x = 100\%$ and all terms). For FireFox, the respective numbers are 12 minutes, 78% top-5 accuracy, 9 times faster and 7% more accurate than the base model (Table 5.2). For FreeDesktop, configured model is 10 times faster than the base model with 3% higher accuracy (Table 5.3).

Tables 5.4, 5.5, and 5.6 shows top-1 and top-5 best accuracy, and the total processing time, respectively for all datasets in Table 2.1 when we ran Bugzie with four types of configurations: base model with all developers and all terms (Column Base), the one with candidate selection (Column C.S.), the one with term selection (Column T.S.), and the one with both (Column Both).

Project	**Base**	**C.S**	**T.S**	**Both**
FireFox	28.0	30.0	32.1	34.6
Eclipse	40.5	40.9	42.6	45.0
Apache	39.8	39.8	39.8	39.8
Netbeans	26.3	26.3	31.8	32.3
FreeDesktop	47.1	47.3	51.2	51.2
Gcc	48.6	48.7	48.6	48.7
Jazz	28.4	28.4	31.3	31.3

Table 5.4: Top-1 Prediction Accuracy (%)

Project	**Base**	**C.S**	**T.S**	**Both**
FireFox	70.7	72.4	73.9	77.8
Eclipse	72.0	72.7	80.1	83.2
Apache	75.0	74.9	75.0	75.0
Netbeans	54.2	59.5	60.4	61.3
FreeDesktop	77.9	78.0	81.1	81.1
Gcc	79.2	79.3	79.2	79.6
Jazz	72.6	72.6	75.3	75.3

Table 5.5: Top-5 Prediction Accuracy (%)

Project	**Base**	**C.S**	**T.S**	**Both**
FireFox	1:50:04	31:24	24:14	12:04
Eclipse	1:39:12	50:47	26:28	12:00
Apache	1:08:23	46:24	1:05:00	36:59
Netbeans	17:04	11:51	4:49	2:30
FreeDesktop	20:35	17:26	3:03	2:07
Gcc	14:37	7:08	11:44	7:08
Jazz	24:45	21:12	1:37	1:37

Table 5.6: Processing Time Comparison

Generally, the top-5 accuracy achieves the best results in the range of 75-83% for all projects (except for NetBeans - 61.3%). That is, approximately in five out of six cases, the correct fixer is in Bugzie's recommending list of five developers. The best results for top-1 accuracy are from 31-51%. That is, in one out of 2-3 cases, the single recommended developer by Bugzie is actually the fixer of the given bug report. Importantly, comparing with the base model, the models with tuned parameters (C.S., T.S., and Both) significantly improve time efficiency, while maintaining the high levels of accuracy. Even in five out of seven systems, tuned parameters help increase top-1 accuracy levels from 3-7% and top-5 ones from 3-11%.

5.5 Comparison Results

This section presents our evaluation result to compare Bugzie with existing state-of-the-art approaches. For the comparison purpose, we used Weka [38] to re-implement the existing state-of-the-art approaches [12, 2, 7, 28] with the same experimental setup and with the descriptions of their approaches in their papers. Cubranic and Murphy [12] use Naive Bayes. Anvik *et al.* [2] employ SVM, Naive Bayes, and C4.5's classifiers. Bhattacharya and Neamtiu [7] use Naive Bayes and Bayesian network with and without incremental learning. We re-implemented Matter *et al.* [28]'s vector-space model (VSM) according to their paper. For comparison, the terms were extracted only from the bug reports.

Because some machine-learning approaches implemented in Weka (e.g. C4.5) can not scale up to the full datasets, we prepared smaller datasets, which have 3-year histories of the full datasets (see Table 5.7). Tables 5.8 and 5.9 show the comparison result in accuracy for the top-1 and top-5 recommendation. Training and prediction time are given in Tables 5.10 and 5.11.

Project	Time	Record	Fixer	Term
Firefox	01-01-2008 to 10-28-2010	77,236	1,682	85,951
Eclipse	01-01-2008 to 10-28-2010	69,829	1,510	103,690
Apache	01-01-2008 to 01-01-2011	28,682	1,354	80,757
NetBeans	01-01-2008 to 11-01-2010	23,522	380	42,797
FreeDesktop	01-01-2008 to 12-05-2010	10,624	161	37,596
Gcc	01-01-2008 to 10-28-2010	6,865	161	20,279
Jazz	06-01-2005 to 06-01-2008	34,228	156	39,771

Table 5.7: 3-Year Fixing History Data

Project	NB	InB	BN	InBN	C4.5	SVM	VSM	Bugzie
Firefox	19.8	21.7	12.9	13.2	24.1	25.7	13.4	29.9
Eclipse	23.7	25.9	12.2	14.1	23.8	27.4	12.2	38.9
Apache	24.3	24.7	11.3	11.6	21.6	26.2	12.0	40.0
NetBeans	16.8	2.7	7.2	5.8	17.9	21.8	8.0	29.2
FreeDesktop	37.1	38.1	31.8	32.6	35.3	42.2	23.2	52.7
Gcc	32.8	33.3	44.2	45.6	39.3	43.0	10.2	45.7
Jazz	19.9	20.4	22.6	22.7	20.5	27.9	6.4	30.0

Table 5.8: Comparison of Top-1 Prediction Accuracy (%)

Project	NB	InB	BN	InBN	C4.5	SVM	VSM	Bugzie
Firefox	43.5	45.8	29.4	30.5	32.6	54.8	33.6	71.8
Eclipse	47.1	49.8	27.9	31.9	33.0	53.0	30.9	71.7
Apache	45.3	46.0	26.6	28.4	32.4	47.6	30.7	78.0
NetBeans	38.5	11.6	21.9	18.9	26.9	45.2	20.8	59.8
FreeDesktop	63.5	65.2	57.2	59.1	47.9	69.0	54.5	80.0
Gcc	71.3	72.5	69.6	71.5	57.5	77.0	37.3	88.8
Jazz	50.3	50.1	55.4	55.8	34.6	67.4	18.9	73.2

Table 5.9: Comparison of Top-5 Prediction Accuracy (%)

As seen, Bugzie consistently outperforms other approaches both in term of prediction accuracy and time efficiency for all subjects. For example, for Eclipse, in term of top-5 accuracy, the second best model is SVM, which has almost 18 hours of processing time and achieves 53% top-5 accuracy, while Bugzie takes only 22 minutes and achieves 72% top-5 accuracy. That is, Bugzie is about 49 times faster and relatively 19% more accurate. In term of processing time, the second best model for Eclipse is VSM, which takes 14 hours and achieves 31% top-5 accuracy, i.e. it is 38 times slower, and 41% less accurate than Bugzie. Generally, ML-based approaches takes from hours to days (even almost a month) to finish training as well as predicting. Bugzie has its training time of tens of minutes to half an hour and prediction time of only seconds, while still achieves higher accuracy.

Decision tree approach (C4.5) has low time efficiency: it takes nearly 28 days for training on Eclipse dataset (with about 70K bug reports). Naive Bayes model takes less time for training (around 9 hours), but much more time for recommending (5.5 days). It is also less accurate than Bugzie: 24% versus 39% (top-1) and 47% versus 72% (top-5). It is similar for Bayesian

Network (15 hours for training and 7.5 days for predicting, with 13% and 28% of top-1 and top-5 accuracy).

Generally, the corresponding accuracy of incremental NB and BN is from 7-21% and 15-38% less than Bugzie for top-1 and top-5 prediction, respectively.

Project	**NB**	**InB**	**BN**	**InBN**	**C4.5**	**SVM**	**VSM**	**Bugzie**
Firefox	9 h	22 h	12 h	33 h	26 d	6 h	42 m	28 m
Eclipse	9 h	37 h	15 h	2 d	28 d	6 h	39 m	21 m
Apache	3 h	8 h	7.5 h	19 h	25 d	2.5 h	1 m	17 m
NetBeans	1 h	4 h	2 h	6 h	10 d	1 h	14 m	10 m
FreeDesktop	18 m	39 m	27 m	1 h	2 d	19 m	13 m	6 m
Gcc	5 m	14 m	8 m	22 m	27 h	9 m	13 m	5 m
Jazz	3 h	4 h	3.5 h	6 h	22 h	4 h	2 m	9 m

Table 5.10: Comparison of Training Time (s: seconds, m: minutes, h: hours, d: days)

Project	**NB**	**InB**	**BN**	**InBN**	**C4.5**	**SVM**	**VSM**	**Bugzie**
Firefox	3 d	3 d	4 d	4.5 d	9 m	8 h	8 h	30 s
Eclipse	5.5 d	5 d	7.5 d	8 d	14 m	12 h	13 h	18 s
Apache	10 h	2 d	25 h	4 d	1 m	48 m	6 h	31 s
NetBeans	14 h	11 h	22 h	15 h	2 m	1 h	1.5 h	5 s
FreeDesktop	4 h	4 h	6 h	5.5 h	48 s	15 m	23 m	3 s
Gcc	40 m	40 m	35 m	25 m	14 s	4 m	8 m	4 s
Jazz	6.5 h	6.5 h	7 h	7 h	10 s	31 m	5 m	5 s

Table 5.11: Comparison of Prediction Time (s: seconds, m: minutes, h: hours, d: days)

5.6 Discussions and Comparisons

Our results suggest that machine learning classification models are less efficient for very large numbers of bug records/fixers. Especially, tree induction models (e.g. C4.5) require all training data to fit in the memory to be efficient [18].

SVM is not well-suited since it is specialized towards classification problems than ranking problems. Using SVM approach, for each developer d, we need to train a classifier SVM_d to distinguish the bug reports that d is able to fix (e.g. $SVM_d(B)) = 1$) and the others (e.g. $SVM_d(B)) = -1$). To adjust to a ranking problem, we need another measure $R_d(B)$ to measure the confidence on the event that d is able to fix B, which is computed as the distance from the vector representing B to the separated hyperplan of SVM_d. Since the classifiers are trained independently, the ranking functions $R_d()$ are not trained competitively together to reflect the actual ranking they should provide (e.g. if both d and d' are considered capable to a bug report B , $R_d(B) > R_{d'}(B)$ might not imply that d is more capable than d' in fixing B). In contrast, Bugzie actually learns/models the ranking functions, i.e. $\mu_t(d)$ and $\mu_B(d)$. Thus, $\mu_B(d) > \mu_B(d')$ does imply that d is more capable than d' in fixing B.

Bayesian models (Bayesian Network) and similarity-based models (e.g. Vector Space Model) can be used for a ranking problem. Using Naive Bayes (NB), given a bug report B as a set of terms, the probability that this bug report belongs to the class of bug reports associated with a developer d is:

$$P(d|B) \propto P(d).P(B|d) = P(d).\prod_{t \in B} P(t|d)$$

In this formula, $P(d)$ is the probability of observing developer d in the fixing data and $P(t|d)$ is the probability of observing term t in the bug reports fixed by d. This formula is used to rank the developers for recommendation.

However, there are two reasons that NB is less suited for automatic bug triaging. First, the probability of assigning developer d to a bug report $P(d|B)$ is proportional to $P(d)$. That is, the more frequently d fixes, the higher chance (s)he is assigned to a new report. This might not fit well with the locality of fixing activity. For example, in practice, there often happens that a developer has been active in bug-fixing for certain technical areas in a period of time, and moves on to other areas. He might have extensive past fixing activities, but does not handle those technical issues anymore. NB still tends to give her/him higher probability due to his past activities. In contrast, Bugzie will not have her/him in its candidate list, if it finds that (s)he has not fixed any bug for a long time.

Second, an important assumption in NB is the independence of the features (i.e. terms), which gives:

$$P(B|d) = \prod_{t \in B} P(t|d)$$

while in bug reports, the terms, especially those relevant to a technical issue, tend to co-occur, i.e. are highly correlated. Let d be a developer with fixing expertise on version control, t = repository and t' = cvs be two terms associated with that concern. t and t' highly co-occur in the bug reports on version control. Assume that, d fixes 100 bug reports, 70 (of 100) containing t, 60 containing t' and 50 containing both of them. Thus, we have $P(t|d) = 0.7$, $P(t'|d) = 0.6$ and $P(t, t'|d) = 0.5$. However, for a bug report B containing both terms, NB will have $P(B|d) = P(t|d) * P(t'|d) = 0.7 * 0.6 = 0.42$, which is likely different from $P(t, t'|d)$. Thus, the feature independence assumption reduces the probability $P(B|d)$. Moreover, that product formula is also sensitive to noises. For example, if B contains t and a misspelled word e, which rarely occurs in bug reports fixed by d ($P(e|d)$ is very small). Then, $P(B|d) = P(t|d) * P(e|d)$ is much smaller than $P(t|d)$ (e.g. $P(e|d) = 0.1$. Then, $P(B|d) = P(t|d) * P(e|d) = 0.7 * 0.1 = 0.07$, much smaller than $P(t|d) = 0.7$).

For Bayesian Network models, the assumption for feature independence is not enforced. However, they still face the same issue, i.e. $P(d|B)$ is proportional to $P(d)$. Thus, BN models are not well suited with the locality of fixing activity.

Vector Space Model (VSM) is IR-based. VSM collects all terms in bug reports into a corpus. It builds the term-fixer matrix in which a fixer is profiled by a vector whose entries equal to the frequencies of the corresponding terms in his fixed bug reports. Developers whose vectors have highest similarity to the vector for a new report are suggested. VSM is less suitable for bug triaging than Bugzie. First, term selection is less flexible because VSM requires all vectors to have the same size. Also, cosine similarity might not be a proper similarity measure of fixing capability because it does not take into account the lengths of vectors in comparison, i.e. a developer's extensive fixing experience could be overlooked.

Here is a simple example. Assume that a system has two aspects: data processing and user interface, with corresponding two terms $t =$ `database` and $t' =$ `gui`. Developer d has fixed 1,000 bug reports on `database` and 500 bug reports on `gui`. Thus, he has a vector-based profile $v =< 1000, 500 >$. Developer d' has fixed only 2 bug on database, thus has profile $v' =< 2, 0 >$. Now, given a bug report B on `database`, which has a representing vector $B =< 1, 0 >$. Then, computing cosine similarity gives $cos(v, B) \approx 0.89$ and $cos(v', B) = 1$. That means, cosine similarity considers d' a better match to B than d, thus, VSM would assign d' to B. However, d should be more capable toward B, given his extensive experience on that aspect.

In contrast, Bugzie takes this into account. We have $n_d = 1,000+500$, $n_t = 1,000+2$, and $n_{d,t} = 1,000$, thus, $\mu_t(d) = 1,000/(1,500 + 1,002 - 1,000) \approx 0.67$. For d', we have $n_{d'} = 2$, $n_{d',t} = 2$, thus $\mu_t(d') = 2/(2+1,002-2) \approx 0.002$. Since B contains only t, $\mu_B(d) = \mu_t(d)$ and $\mu_B(d') = \mu_t(d')$. Therefore, Bugzie assigns d to B, because $\mu_B(d)$ is much higher than $\mu_B(d')$.

In brief, comparing to those models, Bugzie is better suited to bug triaging because it is adapted to the *ranking nature* of the problem, the *locality* of fixing activity, the *co-occurrences* (i.e. dependency) of technical terms associated with the same technical aspect, and the *evolutionary nature* of software development. In addition to significantly higher accuracy, Bugzie also has significantly higher efficiency than existing approaches because 1) training/recommending relies on simple arithmetic calculations on counting values (Chapter 3), 2) updating is fast and truly incremental, and 3) selections of terms and developers reduce processing time.

In Bugzie, technical terms are selected based on their levels of direct association to developers. One could use other feature selection methods such as information theoretic measures (e.g. information gain). Topic-modeling [8] could be used to identify technical topics and associated terms. Also, other developers' selection strategies [32] could be applied.

5.7 Threats to Validity

The re-produced result of existing approaches: We re-implemented existing approaches via Weka [38] and via our own code, rather than using their tools, which are not available. However, our re-implementation was based strictly on the descriptions in their papers. Furthermore, Weka tool might not be always optimized for best time-efficiency.

The correctness of bug database: there might be some bugs are closed and then recurring, i.e. they are not actually/comprehensively fixed by the latest assigned fixer(s).

CHAPTER 6 Related Work and Conclusions

6.1 Related Work

There are several approaches that apply machine learning (ML) and/or information retrieval (IR) to automatic bug triaging. The first approach along that line is from Cubranic and Murphy [12]. The titles, descriptions, and keywords are extracted from bug reports to build a classifier for developers using Naive Bayes technique. The classifier then suggests potential fixers based on the classification of a new bug report. Their prediction accuracy is up to 30% on an Eclipse's bug report data set from Jan to Sep-2002. Anvik *et al.* [2] also follow similar ML approach and improve Cubranic *et al.*'s work by filtering out invalid data such as unfixed bug reports, no-longer-working or inactive developers. With three different classifiers using SVM, Naive Bayes, and C4.5, they achieved a precision of up to 64%. Comparing to those ML approaches, Bugzie has several departure points. First, Bugzie addresses bug triaging as a ranking problem, instead of a classification one. Thus, Bugzie is able to more precisely provide the ranked list of potential fixers, while the outcome of a classifier has the assignment of a bug report to one specific developer. Additional and less accurate ranking scheme was used in their approaches (Section 5.6). Second, simple fuzzy set computation with its counting values (Chapter 3) is much more time efficiency than ML approaches in training/prediction. Importantly, Bugzie's truly incremental learning can further improve efficiency. Third, Bugzie takes into account the co-occurrences of terms for the same technical issue. Finally, taking advantage of the locality of fixing activity and term selection, Bugzie copes well with software evolution and improves its accuracy and efficiency.

Another approach is from Bhattacharya and Neamtiu [7]. They use ML with Bayesian

Network and Naive Bayes. Those models are less precise than Bugzie since they cannot handle co-occurrent technical terms, and suffer other limitations as in ML approaches (Section 5.6). To improve ranking, they utilize *bug tossing graphs*, which represent the re-assignments of a bug to multiple developers before it gets resolved (often called *bug tossing*). As shown, Bugzie outperformed both (incremental) NB and BN from 6-20% and 13-35% for top-1 and top-5 accuracy, respectively. Despite of incremental learning, for NB and BN, their training and prediction time for Eclipse is from 9-15 hours and 5.5-7.5 days, while Bugzie takes only minutes to an hour. Bugzie is also able to support developer and term selections.

The idea of bug tossing graphs was introduced by Jeong *et al.* [21]. Their Markov-based model learns from the past the patterns of bug tossing from developers to others after a bug was assigned, and it uses such knowledge to improve bug triaging. Their goal is more toward reducing the lengths of bug tossing paths, rather than addressing the question of who should fix a given bug as in an initial assignment. We will explore the combination of Bugzie and bug tossing graphs for further improvement.

Lin *et al.* [26] use ML with SVM and C4.5 classifiers on both textual and non-text fields (e.g. bug type, submitter, phase ID, module ID, and priority). Running on a proprietary project with only 2,576 bug records, their models achieve the accuracy of up to 77.64%. The accuracy is 63% if module IDs were not considered. Bugzie has higher accuracy and could integrate non-text fields for further improvement. Podgurski *et al.* [31] utilize ML to classify/prioritize bug reports, but not directly support bug triaging. Di Lucca *et al.* [27] use Bayesian and VSM to classify maintenance requests. Such classification can be used in bug triaging.

Other researchers use IR for automatic bug triaging. Canfora and Cerulo [10, 9] use the terms of fixed change requests to index source files and developers, and query them as a new change request comes for bug triaging. The accuracy was not very good (10-20% on Mozilla and 30-50% on KDE).

Matter *et al.* [28] introduce Develect, a VSM model for developers' expertise by extracting terms in their contributed code. A developer's expertise is represented by a vector of frequencies of terms appearing in her/his source files. The vector for a new bug report is compared with the ones for developers for bug triaging. Testing on 130,769 bug reports in Eclipse, the accuracy is not as high as Bugzie (up to 71% with top-10 recommendation list). Compared to Develect, Bugzie's fuzzy sets first enable more *flexible computation and modeling* of developers' bug-fixing expertise. All vectors in Develect must have the same length. With the fuzzy set nature, Bugzie allows to select a small yet significant set of terms to represent each developer. Second, Develect assumes the independence of features/terms.

Moreover, as *a project evolves*, VSM must *recompute* the entire vector set, while Bugzie incrementally updates its data with high efficiency.

Baysal *et al.* [4] proposed to enhance VSM in modeling developers' expertise with preference elicitation and task allocation. Rahman *et al.* [32] measure the quality of assignment by matching the requested (from bug reports) and available (from developers) competence profiles. For automatic support, they need reverse engineering of developers' *competence profiles* [32]. They start with profiling each bug and developer based on competencies and skills, then they used a greedy search algorithm to find the best suitable developer who has the shortest distance and available within a specified look-ahead time. Their approach is extremely difficult [32].

Other researchers categorize/assess bug reports based on their quality, severity levels, duplications, or relations [5, 34, 36, 33, 19, 20, 29, 6, 25, 15]. Automatic tools were built to predict the fixing time and effort for a bug report [22, 37].

Our preliminary model on Bugzie [35] represents developers' bug-fixing expertise with *all* extracted terms and cannot accommodate well the locality of their fixing activities and software evolution. Thus, it is not well-suited for the evolutionary nature of software development. Moreover, preliminary results were only on Eclipse data with 3-years of development and were not as accurate and efficient as those of the model in this paper. Fuzzy set theory was also used in automatic tagging [1].

6.2 Conclusions

We propose Bugzie, a fuzzy set and cache-based approach for automatic bug triaging. A fuzzy set represents the set of capable developers of fixing the bugs related to a technical term.

The membership score of a developer to such fuzzy set is calculated based on her/his fixed bug reports, and is incrementally updated. Such fuzzy sets are computed for each term in a new bug report and are union'ed to find capable fixers. With flexible caching of developers and terms, Bugzie can accommodate the locality of fixing activity, the co-occurrences of the terms of same technical aspects, and software evolution.

Our evaluation results on large-scale subject systems show that Bugzie achieves significantly higher levels of efficiency and correctness than existing state-of-the-art approaches. For example, it could process the whole Eclipse bug dataset, containing around 178K bug reports and having more than 2,100 active developers, in 12 minutes with 45% and 83% accuracy on top-1 and top-5 recommendations, respectively. That means, in almost half of the cases, the single recommended developer is the actual fixer of the given bug report, and in 83% of the cases, (s)he is in the list of 5 recommended developers.

In 7 subject projects, Bugzie's accuracy for top-1 and top-5 recommendations is generally in the range of 31-51% and 70-83%, respectively. It selects around 10-40% of recent fixers as candidates, and characterizes/profiles each candidate with 3-20 most significant terms. Importantly, while existing approaches take from hours to days (even almost a month) to finish training as well as predicting, in Bugzie, training time is from tens of minutes to an hour, while it still consistently achieves higher accuracy. Bugzie's top-1 and top-5 accuracy levels are higher than those of the second best approach from 4-15% and 6-31%, respectively.

Bibliography

[1] Jafar M. Al-Kofahi, Ahmed Tamrawi, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Fuzzy set approach for automatic tagging in evolving software. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[2] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 361–370, New York, NY, USA, 2006. ACM.

[3] Apache bug tracking system. https://issues.apache.org/jira/.

[4] O. Baysal, M.W. Godfrey, and R. Cohen. A bug you like: A framework for automated assignment of bugs. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pages 297 –298, May 2009.

[5] N. Bettenburg, R. Premraj, T. Zimmermann, and Sunghun Kim. Duplicate bug reports considered harmful...really? In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 337 –345, 10 2008.

[6] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 308–318, New York, NY, USA, 2008. ACM.

[7] Pamela Bhattacharya and Iulian Neamtiu. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In *Proceedings of the 2010 IEEE Inter-*

national Conference on Software Maintenance, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[8] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, March 2003.

[9] Gerardo Canfora and Luigi Cerulo. How software repositories can help in resolving a new change request. In *In Workshop on Empirical Studies in Reverse Engineering*, 2005.

[10] Gerardo Canfora and Luigi Cerulo. Supporting change request assignment in open source development. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 1767–1772, New York, NY, USA, 2006. ACM.

[11] Kevin Crowston and Barbara Scozzi. Coordination practices within floss development teams the bug fixing process. In *In Computer Supported Acitivity Coordination*, pages 21–30. INSTICC Press, 2004.

[12] Davor Cubranic. Automatic bug triage using text categorization. In *In SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, pages 92–97. KSI Press, 2004.

[13] Eclipse bug tracking system. https://bugs.eclipse.org/bugs/.

[14] Firefox bug tracking system. https://bugzilla.mozilla.org/.

[15] Michael Fischer, Martin Pinzger, and Harald Gall. Analyzing and relating bug report data for feature tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering*, WCRE '03, pages 90–, Washington, DC, USA, 2003. IEEE Computer Society.

[16] Freedesktop bug tracking system. https://bugs.freedesktop.org/.

[17] Gcc bug tracking system. http://gcc.gnu.org/bugzilla/.

[18] Jiawei Han. *Data Mining: Concepts and Techniques.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[19] L. Hiew. Assisted detection of duplicate bug reports. Master's thesis, The University of British Columbia, Vancouver, Canada, May 2006.

[20] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 34–43, New York, NY, USA, 2007. ACM.

[21] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 111–120, New York, NY, USA, 2009. ACM.

[22] Sunghun Kim and E. James Whitehead, Jr. How long did it take to fix bugs? In *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, pages 173–174, New York, NY, USA, 2006. ACM.

[23] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 489–498, Washington, DC, USA, 2007. IEEE Computer Society.

[24] G.J. Klir and Bo Yuan. *Fuzzy Sets and Fuzzy Logic: Theory and Applications*. Prentice Hall, 1995.

[25] Andrew J. Ko, Brad A. Myers, and Duen Horng Chau. A linguistic analysis of how people describe software problems. In *Proceedings of the Visual Languages and Human-Centric Computing*, pages 127–134, Washington, DC, USA, 2006. IEEE Computer Society.

[26] Zhongpeng Lin, Fengdi Shu, Ye Yang, Chenyong Hu, and Qing Wang. An empirical study on bug assignment automation using chinese bug data. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 451–455, Washington, DC, USA, 2009. IEEE Computer Society.

[27] G. A. Di Lucca, M. Di Penta, and S. Gradara. An approach to classify software maintenance requests. In *In Proc., International Conference on Software Maintenance (ICSM*, pages 93–102. IEEE Computer Society, 2002.

[28] Dominique Matter, Adrian Kuhn, and Oscar Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pages 131–140, Washington, DC, USA, 2009. IEEE Computer Society.

[29] T. Menzies and A. Marcus. Automated severity assessment of software defect reports. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 346 –355, 10 2008.

[30] Netbeans bug tracking system. http://netbeans.org/bugzilla/.

[31] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 465–475, Washington, DC, USA, 2003. IEEE Computer Society.

[32] Md. Mainur Rahman, Guenther Ruhe, and Thomas Zimmermann. Optimized assignment of developers for fixing bugs an initial evaluation for eclipse projects. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 439–442, Washington, DC, USA, 2009. IEEE Computer Society.

[33] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. Detection of duplicate defect reports using natural language processing. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 499–510, Washington, DC, USA, 2007. IEEE Computer Society.

[34] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the*

32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10, pages 45–54, New York, NY, USA, 2010. ACM.

[35] Ahmed Tamrawi, Tung Thanh Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. Fuzzy set-based automatic bug triaging. ICSE '11 (NIER). ACM (To appear), 2011.

[36] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 461–470, New York, NY, USA, 2008. ACM.

[37] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 1–, Washington, DC, USA, 2007. IEEE Computer Society.

[38] Weka: Data mining software in java. http://www.cs.waikato.ac.nz/ml/weka/.

[39] Bug tracking system. http://en.wikipedia.org/wiki/Bug_tracking_system.

[40] Wvtool: Word vector tool. http://sourceforge.net/projects/wvtool/.