Fuzzy Set and Cache-based Approach for Bug Triaging

Ahmed Tamrawi
atamrawi@iastate.edu

Tung Thanh Nguyen
tung@iastate.edu

Jafar M. Al-Kofahi
jafar@iastate.edu

Tien N. Nguyen
tien@iastate.edu

Electrical and Computer Engineering Department
Iowa State University
Ames, IA 50011, USA

ABSTRACT

Bug triaging aims to assign a bug to the most appropriate fixer. That task is crucial in reducing time and efforts in a bug fixing process. In this paper, we propose Bugzie, a novel approach for automatic bug triaging based on fuzzy set and cache-based modeling of the bug-fixing expertise of developers. Bugzie considers a software system to have multiple technical aspects, each of which is associated with technical terms. For each technical term, it uses a fuzzy set to represent the developers who are capable/competent of fixing the bugs relevant to the corresponding aspect. The fixing correlation of a developer toward a technical term is represented by his/her membership score toward the corresponding fuzzy set. The score is calculated based on the bug reports that (s)he has fixed, and is updated as the newly fixed bug reports are available. For a new bug report, Bugzie combines the fuzzy sets corresponding to its terms and ranks the developers based on their membership scores toward that combined fuzzy set to find the most capable fixers. Our empirical results show that Bugzie achieves significantly higher accuracy and time efficiency than existing state-of-the-art approaches.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management

General Terms

Algorithms, Human Factors, Management, Reliability

Keywords

Bug Triaging, Fuzzy Set, Developers' Expertise

1. INTRODUCTION

Software bugs are inevitable and bug fixing is an essential, yet costly phase during software development. To improve its efficiency and reduce its cost, one should assign a bug report, which describes some technical issue(s), to the right fixer. This process is known as *bug triaging* [3].

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

To support developers in this task, we propose Bugzie, a novel fuzzy set and cache-based approach for automatic bug triaging. 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. 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)/issue(s).*

The key idea of Bugzie is to model the *fixing correlation/association* of developers toward a technical aspect via fuzzy sets [25]. The fixing correlation/association represents the bug-fixing capability/expertise of developers with respect to the technical aspects in a project. To realize that, 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 the set 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 issue(s) corresponding to t. Then, for a new bug report B, the fuzzy set C_B of capable developers toward the technical aspect(s) reported in B is modeled by the union set of all fuzzy sets corresponding to all the 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 *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% of developers having the most recent fixing activities. Thus, we can apply a filtering technique [21] 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

Table 1: Statistics of All Collected Bug Report Data

Project	Time	Report	Fixer	Term
Firefox	04/07/98 - 10/28/10	188,139	3,014	177,028
Eclipse	10/10/01 - 10/28/10	177,637	2,144	193,862
Apache	05/10/02 - 01/01/11	43,162	1,695	110,231
NetBeans	01/01/08 - 11/01/10	23,522	380	42,797
FreeDesktop	01/09/03 - 12/05/10	17,084	374	61,773
Gcc	08/03/99 - 10/28/10	19,430	293	63,013
Jazz	06/01/05 - 06/01/08	34,228	156	39,771

terms for its computation, Bugzie is flexible to use only the terms that are highly correlated with each developer as the most significant terms to represent his/her fixing expertise.

To adapt to software evolution, Bugzie updates its model regularly (e.g. the lists of fixer candidates and terms, and membership scores) as new fixing information is available.

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 about 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.

In summary, the key contributions of this paper include:

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, yet significant set of terms describing developers' bug-fixing expertise.

Section 2 presents an empirical study and a motivating example for Bugzie. Sections 3 and 4 describe our approach and algorithm. Section 5 presents our empirical evaluation. Related work is in Section 6. Conclusions appear last.

2. EMPIRICAL STUDY AND MOTIVATION

This section presents our empirical study and an analysis that motivates our approach.

2.1 Data Collection

Our datasets contain bug reports, corresponding fixers, and related information (e.g. summary, description, and creation/fixing time). Table 1 shows our collected datasets of seven projects: FireFox (FF) [14], Eclipse (EC) [13], Apache (AP) [2], Netbeans (NB) [30], FreeDesktop (FD) [15], Gcc (GC) [17], and Jazz (JZ). 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*. Duplicate and unresolved (open) bug reports were excluded. For the duplicate bug reports, only the master ones were assigned to the fixers. Thus, we kept the master ones and removed all other reports marked as their duplicates. Re-open/un-finished bug fixes were not included either. In Table 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.

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,...

Figure 1: Bug report #6021 in Eclipse project

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 ...

Figure 2: Bug report #0002 in Eclipse project

For each bug report, we extracted its unique bug ID, the actual fixing developer's ID, email address, creating and fixing time, summary, and full description. Comments and discussions are excluded. We merged the summary and description of each bug report, extracted their terms and preprocessed them, such as stemming for term normalization and removing grammatical and stop words. Column Term in Table 1 shows the total numbers of terms in all datasets.

2.2 A Motivating Example

Let us present a motivating example in our collected data. Figure 1 depicts a bug report from Eclipse, with the relevant fields including the fixing developer (AssignedTo), a short summary (Summary), and a description (Description) of the bug. 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, path, etc. 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's bug reports, we found that James also fixed several other VCM-related bugs, for example, bug #0002 (Figure 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.

2.3 Implications and our Approach

This example 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 him to have *bug-fixing expertise/capability* on that aspect, i.e., 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 solution is different from existing approaches to automatic bug triaging [3, 5, 10, 28]. The first line of ideas from existing machine learning (ML)-based approaches [3, 5] 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. The second line 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 his fixed bug reports, and then match a new bug report with such profiles to find the fixer(s). Bugzie is aligned more with the second line of ideas, however, it is centered around the modeling of the correlation/association between two sets: *developers* and *terms*. Thus, we have to address the question of how to make the selections and take into account relevant terms and developers.

As shown in Table 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 word 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 Section 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, 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*: *the recent fixing developers are likely to fix bug reports in the near future*.

Table 2: % of Actual Fixers Having Recent Fixes

Recent	EC	FF	JZ	GC	AP	FD	NB
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%

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 shows the result.

As seen, it is consistent in all systems that $p(x)$ is quite large even at a 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 $x = 100\%$, $p(x)$ might 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.

Implications. The result confirms our hypothesis on the locality of fixing activity. This result enables us to apply the filtering technique for recommender systems [21]: 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.

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). Thus, 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 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 fuzzy set theory [25] to model the *fixing correlation/association between developers and the technical terms*, 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 to model developers' expertise.

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 an associated membership function $\mu_t()$, represents the set of capable fixers toward t, i.e. the developers who have the bug-fixing expertise relevant to the 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 technical issue(s) 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 that 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 with 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 notion as in conventional logic. That is, the boundary of the set of developers who are capable of fixing the bugs relevant to t is fuzzy.

Definition 2 gives us $\mu_t(d) = \frac{1}{\frac{n_t+n_d}{n_{d,t}}-1}$. Thus, Bugzie favors (ranks higher) the developers who have emphasized their 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' has fixed several other technical issues while d emphasizes mostly 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. In general, if $\frac{n_t+n_d}{n_{d,t}}$ is smaller, $\mu_t(d)$ will be higher. Therefore, Bugzie favors the specialists who have fixed more bug reports relevant to t (i.e. large $n_{d,t}$) and have emphasized their fixing more on t (i.e. large $n_{d,t}/n_d$).

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 sub-fuzzy sets with strong membership degrees exist, the stronger the membership toward the combined fuzzy set is. According to [25], the membership score of d with respect to 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 for 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 the 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, e.g., the concern of version control in Eclipse might be associated with the 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 $\mu_{t'}(d)$.

$\mu_B(d)$ is also 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 that d had fixed, d has a 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

Let us discuss our selection strategies for fixers and terms.

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, the locality of fixing activity suggests that the loss in accuracy is acceptable. For example, by selecting $x = 50$%, we can reduce in half the computation time, while losing at most 1-3% of accuracy for all subject systems (lines 50%, 100% of Table 2).

3.3.2 Selection of Descriptive Terms

Bugzie measures the term significance 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 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.

Table 3: Term Selection for Eclipse's developers

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

4. ALGORITHM

This section describes the key algorithm in Bugzie. With two adjustable parameters x (for fixer candidates) and k (for selected term lists), Bugzie operates in three main phases: 1) **initializing**, 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.

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 lists $F(x)$, $T_d(k)$, 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 Bugzie's incremental training efficient. Importantly, it fits well with software evolution. 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 with most tendency for fixing bugs. The lists $T_d(k)$ always consist of the terms having highest association with the developers. Existing bug triaging approaches are not sufficiently flexible to support such caching of developers and terms. In Bugzie, time-sensitive knowledge on developers' fixing activities and important terms during software evolution can be taken into account. In future, other cache replacement strategies as in BugCache [23] could be explored, e.g., caching recent buggy terms and their co-occurring terms.

5. EMPIRICAL EVALUATION

We evaluated Bugzie on our collected datasets (Section 2), some of which were used in prior bug triaging research [3, 28, 5]. We evaluated it with various parameters for developers' and terms' selections, and compared it with state-of-the-art approaches [11, 3, 28, 5]. All experiments were on a Windows Vista, 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 [5]. That is, all extracted bug reports from each bug repository in Table 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 $BR_{1,1}$ in frame 1. After that, we performed updating for our training data with tested bug report $BR_{1,1}$, and started recommending for the next bug report $BR_{1,2}$ in frame 1. After completing frame 1, the updated training data was then used to test frame 2 in the same manner. We repeated this process 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 [5], i.e. the ratio between the number of hits over the total number of prediction cases. 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). Bugzie allows to choose x% of top fixers having most recent fixing activities. We ran it with various values of x%, 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 1.

Figures 3 and 4 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

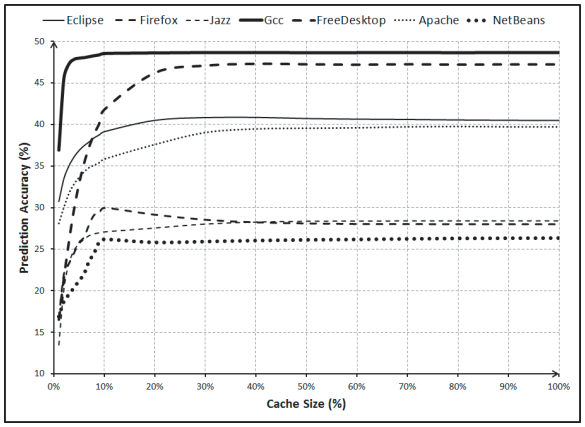

Figure 3: Top-1 Accuracy with Various Cache Sizes

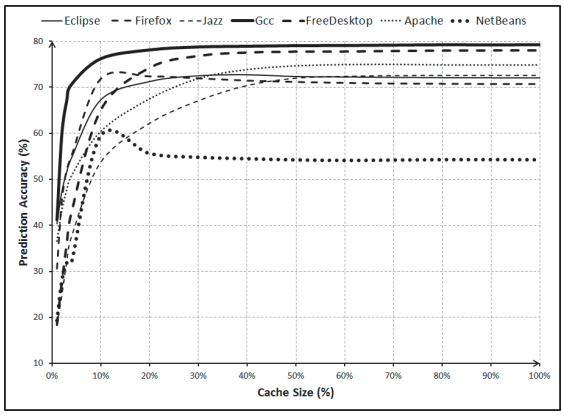

Figure 4: Top-5 Accuracy with Various Cache Sizes

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 displays the total processing 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 larger datasets. However, for FireFox, at $x = 20$%, 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

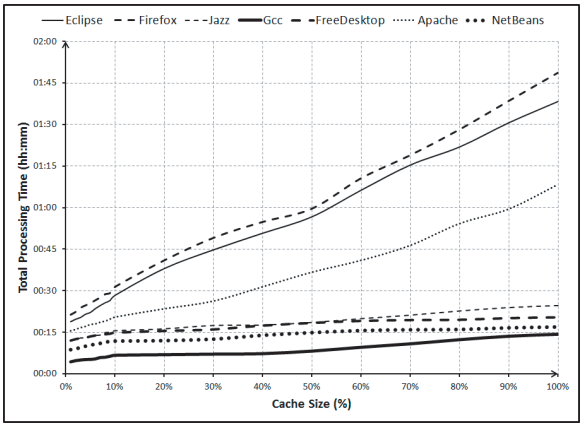

Figure 5: Processing Time with Various Cache Sizes

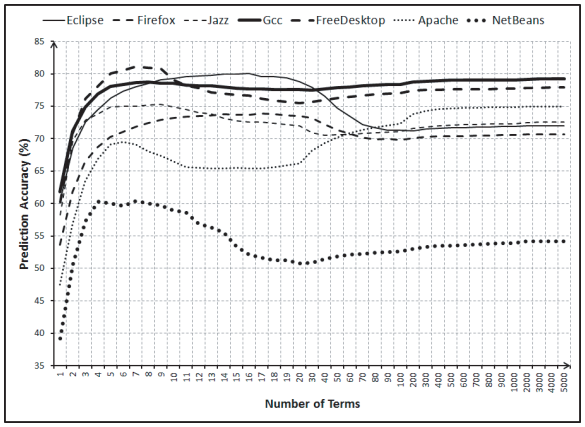

Figure 7: Top-5 Accuracy - Various Term Selection

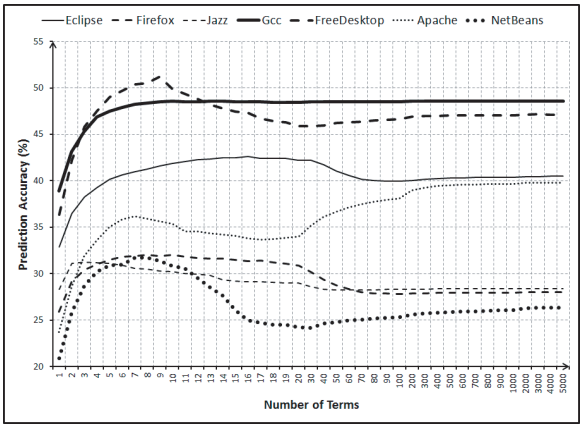

Figure 6: Top-1 Accuracy - Various Term Selection

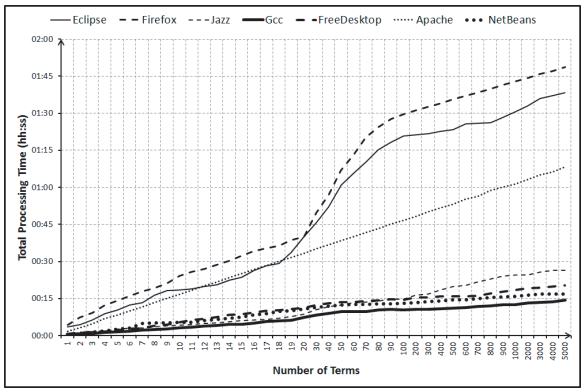

Figure 8: Processing Time - Various Term Selection

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 4, the appropriate size of developer cache depends 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 (Section 3.3.2). 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 $T(k)$ covers most of 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.

Figures 6 and 7 show the results of top-1 and top-5 prediction accuracy on all datasets, with different values of k. As seen, for most projects, 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 $k = 5,000$, 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. 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 significantly improves time efficiency. Figure 8 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 to the cache size of selected terms, showing that Bugzie is scalable well to large projects.

5.4 Selection of Developers and Terms

To evaluate the impacts of both types of selection, we conducted another experiment and tuned the model with dif-

Table 4: Eclipse: Accuracy - Various Parameters

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: FireFox: 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

ferent sizes of developer cache and term cache. Two caches are used together in which Bugzie computes the scores and ranks only the developers in the developer cache for recommendation, and uses only the terms in the term cache for its computation (Section 3.3). We ran Bugzie on all datasets in Table 1 with all combinations of the best values we discovered in the previous experiments as the model's parameters. Tables 4 and 5 show the accuracy and total processing time (hh:mm:ss) with different parameters for Eclipse and FireFox.

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 about 178K bug records and 2K developers) in only 12 minutes and achieves 83% top-5 prediction accuracy. That is about 8.3 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).

Table 6 shows top-1 and top-5 best accuracy when we ran Bugzie with 4 types of configurations: base model with all developers and all terms (Column Base), base model with candidate selection (Column C.S.), base model with term selection (Column T.S.), and base model with both (Column Both).

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 5 out of 6 cases, the correct fixer is in Bugzie's recommending list of 5 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 5 out of 7 systems, tuned parameters help increase top-1 accuracy levels from 3-7% and top-5 ones from 3-11%.

5.5 Comparison Results

For comparison, we used Weka [38] to re-implement existing approaches [11, 3, 5, 28] following the descriptions in their papers. Cubranic and Murphy [11] use Naive Bayes. Anvik *et al.* [3] use Naive Bayes, SVM, and C4.5 classifiers. Bhattacharya and Neamtiu [5] use Naive Bayes and Bayesian Network with and without incremental learning. We re-implemented Matter *et al.* [28]'s vector-space model (VSM) with the terms being extracted only from the bug reports.

Table 7: 3-Year Fixing History Data

Project	Time	Record	Fixer	Term
Firefox	01/01/08 - 10/28/10	77,236	1,682	85,951
Eclipse	01/01/08 - 10/28/10	69,829	1,510	103,690
Apache	01/01/08 - 01/01/11	28,682	1,354	80,757
NetBeans	01/01/08 - 11/01/10	23,522	380	42,797
FreeDesktop	01/01/08 - 12/05/10	10,624	161	37,596
Gcc	01/01/08 - 10/28/10	6,865	161	20,279
Jazz	06/01/05 - 06/01/08	34,228	156	39,771

Since some machine-learning approaches (e.g. C4.5) realized in Weka can not scale up to the full datasets, we prepared smaller datasets, which have 3-year histories of the full ones (Table 7). Table 8 shows the comparison result for top-rank accuracy. Table 9 shows training and prediction time.

As seen, Bugzie consistently outperforms other approaches in term of both 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 (NB) 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 (BN) (15 hours for training and 7.5 days for predicting, with 13% and 28% of top-1 and top-5 accuracy). Generally, the respective accuracy of incremental NB and BN is from 7-21% and 15-38% less than Bugzie's accuracy for top-1 and top-5 prediction.

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].

Using SVM, for a developer d, one needs to train a *classifier* SVM_d to distinguish the bug reports that d is able to fix (i.e. $SVM_d(B) = 1$) from the others (i.e. $SVM_d(B) = -1$). Using it to rank developers, one needs 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 separate 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 viewed as capable to fix 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 models/learns 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.

Table 6: Top-1 & Top-5 Prediction Accuracy (%) - C.S: Fixer Candidate Selection, T.S: Term Selection

	Top-1				Top-5				Time			
Project	Base	C.S	T.S	Both	Base	C.S.	T.S.	Both	Base	C.S.	T.S.	Both
FireFox	28.0	30.0	32.1	34.6	70.7	72.4	73.9	77.8	1:50:04	31:24	24:14	12:04
Eclipse	40.5	40.9	42.6	45.0	72.0	72.7	80.1	83.2	1:39:12	50:47	26:28	12:00
Apache	39.8	39.8	39.8	39.8	75.0	74.9	75.0	75.0	1:08:23	46:24	1:05:00	36:59
Netbeans	26.3	26.3	31.8	32.3	54.2	59.5	60.4	61.3	17:04	11:51	4:49	2:30
FreeDesktop	47.1	47.3	51.2	51.2	77.9	78.0	81.1	81.1	20:35	17:26	3:03	2:07
Gcc	48.6	48.7	48.6	48.7	79.2	79.3	79.2	79.6	14:37	7:08	11:44	7:08
Jazz	28.4	28.4	31.3	31.3	72.6	72.6	75.3	75.3	24:45	21:12	1:37	1:37

In 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 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 him higher probability due to his past activities. In contrast, Bugzie will not have him in its candidate list, if it finds that 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. However, for a bug report B containing both terms, NB will have $P(B|d) = P(t|d) * P(t'|d)$, 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 $P(t|d)$.

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.

In Vector Space Model (VSM), all terms in bug reports are collected 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 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. Moreover, all terms in a bug report are assumed to be independent.

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 (Section 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 [9] could be used to identify technical topics and associated terms. Also, other developers' selection strategies [32] could be applied.

Threats to Validity. Since the tools of existing approaches are not available, we re-implemented them via Weka [38]. However, our code is strictly based on their descriptions.

6. RELATED WORK

There exist machine learning (ML) and information retrieval (IR) approaches to automatic bug triaging. In Cubranic and Murphy [11]'s, the titles, descriptions, and keywords are extracted from the bug reports to build a classifier for developers using Naive Bayes. The classifier suggests potential fixers based on the classification of a new report. Its prediction accuracy is up to 30% on an Eclipse's bug dataset from Jan to Sep-2002. Anvik *et al.* [3] follow similar ML approach and improve Cubranic *et al.*'s work by filtering out unfixed bug reports and inactive developers. With 3 different classifiers using SVM, Naive Bayes, and C4.5, they achieve a precision of up to 64%. Comparing to those ML approaches, Bugzie has some advances. First, it is able to provide more precisely the ranked list of potential fixers. The outcome of a classifier has the assignment of a bug report to one specific fixer, thus, an additional ranking scheme was needed with the classifier. Second, simple fuzzy set computation with its counting values is much more time efficiency than ML approaches in training/prediction. Importantly, Bugzie's truly incremental training and caching fits well with software evolution and improves its performance. Finally, it can handle the co-occurrences of terms of the same technical issue.

Another approach is from Bhattacharya and Neamtiu [5]. They use 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. As shown, Bugzie outperformed (incremental) NB and BN from

Table 8: Comparison of Top-1 and Top-5 Prediction Accuracy (%)

	Top-1								Top-5							
Project	NB	InB	BN	InBN	C4.5	SVM	VSM	Bugzie	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	43.5	45.8	29.4	30.5	32.6	54.8	33.6	71.8
Eclipse	23.7	25.9	12.2	14.1	23.8	27.4	12.2	38.9	47.1	49.8	27.9	31.9	33.0	53.0	30.9	71.7
Apache	24.3	24.7	11.3	11.6	21.6	26.2	12.0	40.0	45.3	46.0	26.6	28.4	32.4	47.6	30.7	78.0
NetBeans	16.8	2.7	7.2	5.8	17.9	21.8	8.0	29.2	38.5	11.6	21.9	18.9	26.9	45.2	20.8	59.8
FreeDesktop	37.1	38.1	31.8	32.6	35.3	42.2	23.2	52.7	63.5	65.2	57.2	59.1	47.9	69.0	54.5	80.0
Gcc	32.8	33.3	44.2	45.6	39.3	43.0	10.2	45.7	71.3	72.5	69.6	71.5	57.5	77.0	37.3	88.8
Jazz	19.9	20.4	22.6	22.7	20.5	27.9	6.4	30.0	50.3	50.1	55.4	55.8	34.6	67.4	18.9	73.2

Table 9: Comparison of Processing Time (s: seconds, m: minutes, h: hours, d: days)

	Train Time								Pred. Time							
Project	NB	InB	BN	InBN	C4.5	SVM	VSM	Bugzie	NB	InB	BN	InBN	C4.5	SVM	VSM	Bugzie
FF	9 h	22 h	12 h	33 h	26 d	6 h	42 m	28 m	3 d	3 d	4 d	4.5 d	9 m	8 h	8 h	30 s
EC	9 h	37 h	15 h	2 d	28 d	6 h	39 m	21 m	5.5 d	5 d	7.5 d	8 d	14 m	12 h	13 h	18 s
AP	3 h	8 h	7.5 h	19 h	25 d	2.5 h	1 m	17 m	10 h	2 d	25 h	4 d	1 m	48 m	6 h	31 s
NB	1 h	4 h	2 h	6 h	10 d	1 h	14 m	10 m	14 h	11 h	22 h	15 h	2 m	1 h	1.5 h	5 s
FD	18 m	39 m	27 m	1 h	2 d	19 m	13 m	6 m	4 h	4 h	6 h	5.5 h	48 s	15 m	23 m	3 s
GC	5 m	14 m	8 m	22 m	27 h	9 m	13 m	5 m	40 m	40 m	35 m	25 m	14 s	4 m	8 m	4 s
JZ	3 h	4 h	3.5 h	6 h	22 h	4 h	2 m	9 m	6.5 h	6.5 h	7 h	7 h	10 s	31 m	5 m	5 s

6-20% and 13-35% for top-1 and top-5 accuracy, respectively. Despite of incremental learning, the training and prediction time of those models for Eclipse is from 9-15 hours and 5.5-7.5 days, while Bugzie takes only minutes to an hour.

The idea of bug tossing graphs was introduced by Jeong *et al.* [22]. Their Markov-based model learns from the past the patterns of bug tossing from developers to others after a bug was assigned. 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.

Lin *et al.* [27] 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 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 can 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.* use Bayesian and VSM to classify maintenance requests [12].

Other researchers use IR for automatic bug triaging. Canfora and Cerulo [10, 8] 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).

In Develect [28], VSM is used to model a developer's expertise by a vector of frequencies of the terms extracted from his contributed source code. The vector for a new bug report is compared with the vectors 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 fuzzy set nature, Bugzie allows to select a small yet significant set of terms for 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 and available competence profiles from bug reports and developers.

Other researchers categorize/assess bug reports based on their quality, severity levels, duplications, fixing time, and relations [7, 34, 36, 33, 19, 20, 29, 6, 26, 16, 24, 37].

Our preliminary Bugzie [35] takes into account all extracted terms and developers in a project's history. Taking advantage of our new finding on the locality of fixing activity and the use of two caching techniques, new Bugzie greatly improves over the old model: reducing processing time about 10 times, and improving up to 12% in top-5 prediction accuracy (Table 6). The improvement is confirmed in our new comprehensive, empirical evaluation on 7 large-scale, long-lived projects, while the preliminary results were only on 3-year Eclipse data. TagRec [1] uses fuzzy set theory to recommend the most relevant software tags (i.e. terms) to software artifacts. It is based on the term-to-term correlation, i.e. modeling the association among terms, which belong to the same space. In contrast, Bugzie is based on the developer-to-term correlation, i.e. modeling the association of developers and terms, which are the entities in different spaces. Caching also helps it fit with bug triaging problem.

7. CONCLUSIONS

We propose Bugzie, a fuzzy set and cache-based approach for automatic bug triaging. A fuzzy set represents the capable developers of fixing the bugs related to a technical issue. The membership score of a developer to a fuzzy set is calculated based on his fixed bug reports, and is incrementally updated. 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 shows that it achieves higher accuracy and efficiency than existing approaches.

Acknowledgment. This project is funded in part by NSF CCF-1018600 grant.

8. REFERENCES

[1] J. Al-Kofahi, A. Tamrawi, T. Nguyen, H. Nguyen, and T.N. Nguyen. Fuzzy Set Approach for Automatic Tagging in Evolving Software. In ICSM'10. IEEE CS, 2010.

[2] Apache. https://issues.apache.org/jira/

[3] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In ICSE '06, pages 361–370. ACM Press, 2006.

[4] O. Baysal, M. W. Godfrey, and R. Cohen. A bug you like: A framework for automated assignment of bugs. In ICPC'09, pages 297-298. IEEE CS, 2009.

[5] P. Bhattacharya and I. Neamtiu. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In *ICSM'10*. IEEE CS, 2010.

[6] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, T. Zimmermann. What makes a good bug report? In FSE'08, pages 308-318. ACM Press, 2008.

[7] N. Bettenburg, R. Premraj, T. Zimmermann, S. Kim. Duplicate bug reports considered harmful... really? In ICSM' 08, pages 337-345. IEEE CS, 2008.

[8] G. Canfora and L. Cerulo. How software repositories can help in resolving a new change request. In *Workshop on Empirical Studies in Reverse Eng.*, 2005.

[9] D. Blei, A.Y. Ng, and M. Jordan. Latent Dirichlet Allocation. J. Mach. Learn. Res. 3 (2003), 993-1022.

[10] G. Canfora and L. Cerulo. Supporting change request assignment in open source development. In SAC'06: ACM symposium on Applied computing, pages 1767–1772. ACM Press, 2006.

[11] D. Cubranic and G. Murphy. Automatic bug triage using text categorization. In SEKE'04: the 16th International Conference on Software Engineering and Knowledge Engineering, pages 92–97. KSI Press, 2004.

[12] G. Di-Lucca, M. Di-Penta, and S. Gradara. An approach to classify software maintenance requests. In ICSM'02, pages 93-102, IEEE CS, 2002.

[13] Eclipse. https://bugs.eclipse.org/bugs/

[14] FireFox. https://bugzilla.mozilla.org/

[15] FreeDesktop. https://bugs.freedesktop.org/

[16] M. Fischer, M. Pinzger, H. Gall. Analyzing and relating bug report data for feature tracking. In WCRE'03, pages 90-99. IEEE CS, 2003.

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

[18] J. Han, M. Kamber, and J. Pei. Data Mining: Concepts and Techniques. The Morgan Kaufmann Series in Data Management Systems, 2006.

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

[20] P. Hooimeijer, W. Weimer. Modeling bug report quality. In ASE '07, pages 34–43. ACM Press, 2007.

[21] D. Jannach, M. Zanker, A. Felfernig, G. Friedrich. Recommender Systems: An Introduction. Cambridge University Press, 2011.

[22] G. Jeong, S. Kim, T. Zimmermann. Improving bug triage with bug tossing graphs. In *FSE'09*, pages 111-120. ACM Press, 2009.

[23] S. Kim, T. Zimmermann, J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *ICSE'07*, pages 489-498. IEEE CS, 2007.

[24] S. Kim, E.J. Whitehead, Jr. How long did it take to fix bugs? In MSR '06, pages 173-174. ACM Press, 2006.

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

[26] A.J. Ko, A.B. Myers, D.H. Chau. A linguistic analysis of how people describe software problems. In VLHCC'06, pagess 127-134. IEEE CS, 2006.

[27] Z. Lin, F. Shu, Y. Yang, C. Hu, and Q. Wang. An empirical study on bug assignment automation using Chinese bug data. In *ESEM'09*, pages 451–455. IEEE CS, 2009.

[28] D. Matter, A. Kuhn, and O. Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *MSR'09*, pp. 131–140. IEEE CS, 2009.

[29] T. Menzies, A.Marcus. Automated severity assessment of software defect reports. In ICSM'08, pages 346–355. IEEE CS, 2008.

[30] Netbeans. http://netbeans.org/bugzilla/

[31] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, B. Wang. Automated support for classifying software failure reports. In ICSE'03, pages 465-475. IEEE CS, 2003.

[32] M. Rahman, G. Ruhe, T. Zimmermann. Optimized assignment of developers for fixing bugs an initial evaluation for Eclipse projects. In ESEM'09, pages 439-442. IEEE CS, 2009.

[33] P. Runeson, M. Alexandersson, O. Nyholm. Detection of duplicate defect reports using natural language processing. In ICSE '07, pp. 499–510. IEEE CS, 2007.

[34] C. Sun, D. Lo, X. Wang, J. Jiang, S.C. Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In ICSE '10, pages 45-54. ACM Press, 2010.

[35] A. Tamrawi, T. Nguyen, J. Al-Kofahi, and T.N. Nguyen. Fuzzy Set-based Automatic Bug Triaging. In ICSE'11 (NIER). ACM, 2011.

[36] X. Wang, L. Zhang, T. Xie, J. Anvik, J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In ICSE'08, pages 461-470. ACM Press, 2008.

[37] C. Weiss, R. Premraj, T. Zimmermann, A. Zeller. How long will it take to fix this bug? In MSR'07, IEEE CS, 2007.

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