FixerCache: Unsupervised Caching Active Developers for Diverse Bug Triage

Song Wang[ϕ], Wen Zhang[ϕǂψ], Qing Wang[ϕψ]
[ϕ]Institute of Software, Chinese Academy of Sciences
[ǂ]State Key Laboratory of Software Engineering of Wuhan University
[ψ]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
{wangsong, zhangwen, wq}@nfs.iscas.ac.cn

ABSTRACT

Context: Bug triage aims to recommend appropriate developers for new bugs in order to reduce time and effort in bug resolution. Most previous approaches for bug triage are *supervised*. Before recommending developers, these approaches need to learn developers' bug-fix preferences via building and training models using text-information of developers' historical bug reports.

Goal: In this paper, we empirically address three limitations of *supervised* bug triage approaches and propose FixerCache, an *unsupervised* approach for bug triage by caching developers based on their activeness in components of products.

Method: In FixerCache, each component of a product has a dynamic *developer cache* which contains prioritized developers according to developers' *activeness scores*. Given a new bug report, FixerCache recommends fixers with high activeness in *developer cache* to participate in fixing the new bug.

Results: Results of experiments on four products from Eclipse and Mozilla show that FixerCache outperforms *supervised* bug triage approaches in both prediction *accuracy* and *diversity*. And it can achieve prediction *accuracy* up to 96.32% and *diversity* up to 91.67%, with top-10 recommendation list.

Conclusions: FixerCache recommends fixers for new bugs based on developers' activeness in components of products with high prediction *accuracy* and *diversity*. Moreover, since FixerCache does not need to learn developers' bug-fix preferences through complex and time consuming processes, it could reduce bug triage time from hours of *supervised* approaches to seconds.

Categories and Subject Descriptors

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

General Terms

Performance, Reliability, Measurement, Human Factors

Keywords

Bug triage, developers' activeness, developers' preferences

ESEM'14, September 18–19, 2014, Torino, Italy.

1. INTRODUCTION

Bug triage is a widely known problem during software development and maintenance, which aims to recommend potential developers for new bugs [1]. Usually, a bug report is reported by a developer and recorded in a bug tracking system, e.g., Bugzilla and JIRA. Traditionally, a developer (also called triager) manually assigns new bug reports to potential developers [21-23]. In order to reduce time and labor for bug triage, many approaches have been proposed to semi-automatically recommend fixers for a new bug, e.g., using machine learning techniques [1,2,13], information retrieval [6,8,10,16], and network analysis [3,9,11,28,29]. These approaches collect historical software bug reports to build and train models, then produce a ranked list of recommended assignees. Since these bug triage approaches need to learn each individual developer's bug-fix preference via text-information-based expertise before triaging bugs, we refer to such approaches as *supervised* approaches.

Although most of *supervised* approaches have been found to be highly accurate and most could achieve prediction *accuracy* range from 60% to 80%, they are not without any flaws. Obviously, some of these approaches consume hours even days [6] to collect and filter data, build and train models. Moreover, text information of bug reports is commonly noisy [17,18,20]. Further, through our pilot studies, we find more of their weaknesses.

First, along with the development of a project, we reveal in our pilot study (see Section2.2) that the more bugs developers fix, the greater text-information-based similarities are among them. This fact affects the discriminative power of these *supervised* approaches which primarily leverage text-information-based expertise to represent a developer's capacity in fixing bugs.

Second, in order to collect sufficient text-information, most of these approaches [1,3-6,8-11,16,28,29] filtered less active developers (developers who fixed less than a certain number of bugs). Usually, only 10%-40% developers and about 40% fixed bug reports are kept after filtering. For instance, bug triage approaches proposed in [1,3-5,9,28,29] removed developers who had fixed less than 50 bugs, after filtering only 30% of all developers who fixed bugs and less than 40% of all fixed bug reports were kept. Our pilot study (see Section2.3) argues that filtering process dramatically lowers the *diversity* of developer recommendation of bug triage, because only developers who have fixed enough bugs could be recommended to fix bugs using these *supervised* bug triage approaches.

Third, similarity-based bug triage policy of these *supervised* bug triage approaches does not take developers' actual bug-fix scope into account. In our pilot study (see Section2.4), we find that most developers only work on one or two components in a product. Moreover, about 95% of their bug-fix records are in preferred components. When *supervised* approaches recommend a developer to fix a new bug with high similarity of text-information, they do not consider whether the developer has worked on the component of the bug in the past.

The above weaknesses of *supervised* bug triage approaches make them less practical and inefficient. Inspired by results of our pilot study (see Section2.4) that "*most developers work on only one or two components in a product*", in this paper, we propose FixerCache, a lightweight approach for bug triage by caching developers based on their *activeness scores* in components of products. We refer to FixerCache as an *unsupervised* bug triage approach, because different from existing *supervised* bug triage approaches, before recommending developers, FixerCache does not need to learn developers' bug-fix preferences through time and labor consuming processes, i.e., extracting and filtering developers and their text information from historical bug reports, building and training model with developers' text-information-based expertise to obtain developers' bug-fix preferences.

Results of experiments on four products of Eclipse and Mozilla projects show that FixerCache can achieve prediction *accuracy* up to 96.32% and *diversity* up to 91.67% with top-10 recommendation list. Moreover, FixerCache could reduce bug triage time from hours of *supervised* approaches to seconds. Our contributions can be summarized as follows:

1. We empirically address three limitations of *supervised* bug triage approaches, which make them less practical and inefficient.

2. We propose FixerCache, an *unsupervised* bug triage approach based on developers' activeness in components of products, which is more efficient and accurate than existing state-of-the-art bug triage approaches.

3. We propose to use both *diversity* and *accuracy* metrics for evaluating performance of bug triage. To our knowledge, we are the first to leverage these two metrics to evaluate the performance of a bug triage approach.

The rest of this paper is organized as follows. Section 2 presents our pilot studies and motivation. Section 3 describes the methodology of our proposed FixerCache. Section 4 evaluates the effectiveness of our bug triage method. Section 5 discusses our approach. Section 6 states the threats. Section 7 presents the related work. Section 8 concludes this paper.

2. PILOT STUDIES AND MOTIVATION

This section reports our motivation and the results of pilot studies on the efficiency validation of *supervised* bug triage approaches.

2.1 Motivation

Our motivation derived from the results of our pilot studies. In our pilot studies, we address three limitations of *supervised* bug triage approaches through three **RQ**s. Further, result of **RQ3** that "*most developers work on only one or two components in a product*" inspires us to propose an *unsupervised* bug triage approach by caching developers based on their activeness in components of products.

Table 1. Details of eight active developers in Eclipse

Project & Period	Developer	# Fixed bugs
Eclipse JDT (2002/01/01-2009/01/01)	Olivier_Thomann (D_{OT})	1242
	Jerome_Lanneluc (D_{JL})	996
	Frederic_Fusier (D_{FF})	663
	Philipe_Mulet (D_{PM})	998
Eclipse Platform (2002/01/01-2009/01/01)	Felipe_Heidrich (D_{FH})	748
	Grant_Gayed (D_{GG})	717
	Silenio_Quarti (D_{SQ})	1138
	Steve_Northover (D_{SN})	880

2.2 Text-information-based expertise of developers

We first try to answer the following research question related to developers' text-information-based expertise used in *supervised* bug triage approaches.

RQ1: *Does the text-information-based expertise discriminate developers efficiently?*

Motivation: Most previous bug triage approaches leverage text-information-based expertise to represent a developer's capacity in fixing a given bug and treat each developer as a label of bug triage classifiers. Usually they use machine learning methods, such as Support Vector Machine (SVM) and Naïve Bayes (NB) to triage a new bug based on the likelihood between the text-information of the new bug and the text-information-based expertise of candidate developers.

Using SVM or NB based bug triage approaches, for each developer d, one need to train a classifier C_d to distinguish bug reports that d has capacity to fix based on text information of bug reports that d has fixed. Thus, the greater the similarity between two classes of a classifier is, the less discriminative power the classifier achieves. We explore the similarity of text-information-based expertise among active developers and examine whether it can efficiently distinguish developers or not.

Approach: We collect fixed bug reports of eight top active developers from two large open source projects: Eclipse JDT and Eclipse Platform (for each project, we select four top active developers), the bugs were reported from 2002/01/01 to 2009/01/01. All of the eight developers had fixed more than 600 bug reports, and details of selected developers are presented in Table 1. Following the techniques described in [1,3-5], we employ *tf* [24], stop words, and stemming to extract string vectors from the summary and description of a bug report. For each developer, we leverage Vector Space Model (VSM) to represent his/her text-information-based expertise and use *Cosine Similarity* [1] to calculate the similarity between two developers.

Given two developers d_x and d_y, we denote $Sim(X,Y)$ as the similarity between them, X and Y are the string vectors of developer d_x and d_y, respectively. $Sim(X,Y)$ is calculated as follows in equation 1.

[1] http://en.wikipedia.org/wiki/Cosine_similarity

Table 2. Similarity between two developers (%)

	D_{JL}	D_{FF}	D_{PM}	D_{FH}	D_{GG}	D_{SQ}	D_{SN}
D_{OT}	45.52	49.53	52.53	29.21	27.64	30.22	28.23
D_{JL}	-	49.29	46.80	29.00	29.30	31.08	29.44
D_{FF}		-	46.09	28.72	28.74	28.62	28.23
D_{PM}			-	26.33	26.32	29.45	26.57
D_{FH}				-	50.02	53.81	53.72
D_{GG}					-	54.37	53.13
D_{SQ}						-	**57.23**

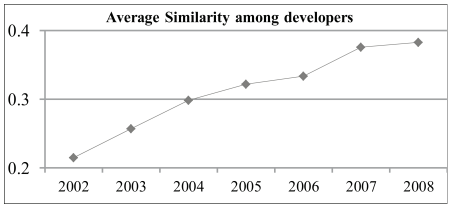


Figure 1. Average similarity among developers

$$Sim(\mathrm{X},\mathrm{Y}) = \frac{X \cdot Y}{\|X\|\|Y\|} = \frac{\sum_{i=1}^{n} X_i \times Y_i}{\sqrt{\sum_{i=1}^{n}(\mathrm{X}_i)^2} \times \sqrt{\sum_{i=1}^{n}(\mathrm{Y}_i)^2}} \quad (1)$$

Results: Table 2 shows the results of our experiment. We find that the similarity among each pair of the eight developers is above 26.00%, and the maximum similarity is up to 57.23%, which between developer Silenio_Quarti and developer Steve_Northover. Moreover, similarity between two developers from the same project is bigger than two developers from different projects, e.g., developers Felipe_Heidrich, Grant_Gayed, Silenio_Quarti and Steve_Northover mainly fix bugs in product Platform and the similarity between any pair of them ranges from 50% to 57.23%. In Table 2, data in the dashed box show the similarity between two developers from different projects and all are bigger than 26%, which are also significant.

We further examine the tendency of average similarity between each pair of the eight developers from Jan. 2002 to Jan. 2009. Results are shown in Figure 1. As we can see that the average similarity among developers is increasing along with the number of bugs developers fixed.

The texts used to characterize developers' fixing preferences are very similar to each other. Their similarities increase along with the number of bugs developers fixed. This lowers the discriminative power of supervised bug triage approaches.

2.3 Filtering developers who fixed less bugs

We further explore the following research question about the influence of filtering developers on performance of *supervised* bug triage approaches.

RQ2: *To what extend does filtering process influence the diversity of developer recommendation of bug triage?*

Motivation: Filtering less active developers (developers who fixed a small number of bug reports) is a widely used process in *supervised* bug triage approaches for collecting sufficient text-information-based expertise to build a classifier with high discriminative power. For example, existing approaches [1,3-6,9] removed developers who fixed bug reports less than 50 in their experiments; existing approaches [8] only selected about 10%-40% of all the developers who had fixed bugs. While in real world bug fixing practices, many of filtered developers have actual bug-fix activities, and using *supervised* bug triage approaches about 60% developers would never be recommended to fix bugs.

Diversity is a widely used metric in recommender system [15,27,34], to recommend more different individuals or avoid monotonous recommendation. The *diversity* of a bug triage approach represents its capacity of modelling real bug fixing practice during software development. A low *diversity* bug triage approach might recommend a small part of developers who have fixed a large number of bugs. For example, assuming ten developers work on a project, and developer *d* fixes more than 50% bugs of the project, if we only recommend *d* for all bug reports in the test dataset, the *accruacy* would be no less than 50%. However, this is not practical and does not accord with real bug resolution practices. In real open source software community, core developers (like developer *d* in the above example), and co-developers (developers who fixed less bugs) [12,35] work together on software projects [28,29]. Assigning bugs only to core developers would mount their workloads and cause more tossing (core developers may reassign bugs to co-developers) [4], thus may reduce the efficiency of bug resolution.

In this work, *diversity* is the ratio of recommended developers in all the unique developers who fixed bugs in a project, and calculated as follows in equation 2:

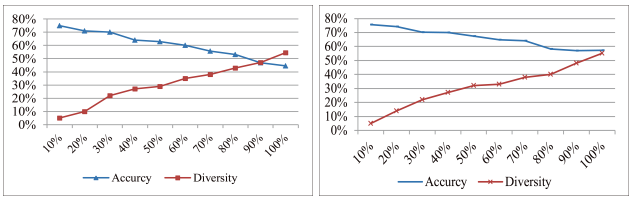
$$Diversity = \frac{\#recommended\ developers}{\#all\ developers\ who\ have\ fixed\ bugs} \quad (2)$$

Filtering less active developers indeed can improve the prediction *accuracy* of recommendation [24]. However, it also lowers the *diversity* of developer recommendation [25,26]. Our focus is on exploring to what extend filtering process lowers the *diversity* of developer recommendation of bug triage.

Approach: We collect all the developers who had fixed bug reports in project Eclipse Platform (238 developers and 32777 bug reports) and project Eclipse JDT (97 developers and 17937 bug reports) from 2002/01/01 to 2009/01/01. Then, we empirically study the influence of filtering process on the *diversity* of developer recommendation with a widely used machine learning based bug triage approach, i.e., SVM. In our experiment, we select *x*% of developers as the dataset, and vary *x* from 10 to 100. For each value of *x*, we build and run SVM-based bug triage approach. Following existing work [6,13], we employ incremental learning to evaluate the result, we sort bug reports in chronological order and divide them into 11 folds and execute 10 rounds to calculate the average top-5 developer prediction *accuracy* and *diversity*.

Results: Figure 2 shows the top-5 prediction *accuracy* and *diversity* for different values of *x*. As seen, the *accuracy* decreases with the increase of *x*, when *x*% > 60%, the average top-5 *accuracy* of SVM is less than 60%. Meanwhile the *diversity* increases along with the increase of *x*. However, even though *x*% = 100%, the average *diversity* of top-5 developer recommendation is only around 50%, which means about 50% developers will never be recommended as bug fixing candidates using a *supervised* bug triage approach, e.g., SVM in this work.

Filtering less active developers in bug triage dramatically lowers the diversity of developer recommendation.



(a) Results of Eclipse Platform (b) Results of Eclipse JDT

Figure 2. Top-5 *accuracy* and *diversity*

Table 3. Distribution of developers' bug-fix activities ("comp" is the abbreviation of "component")

Developer	#fixed bugs in the preferred comp	#fixed bugs in other comp	#comp
D_{OT}	1227	15	5
D_{JL}	994	2	3
D_{FF}	661	2	3
D_{PM}	996	2	3
D_{FH}	748	1	2
D_{GG}	716	1	2
D_{SQ}	1136	2	2
D_{SN}	878	2	3

2.4 Developers' fixing activities scope

We are also interested in answering the following research question about the actual scope of developers' bug-fix activities.

RQ3: *Does a developer has some obvious preferences on some components in a product when he/she fixes bugs?*

Motivation: *Supervised* bug triage approaches triage bug reports primarily based on the likelihood of text-information-based expertise between bug reports and developers, and do not take neither the distribution of bug reports nor developers' bug-fix activities scope in a project into account, which means they assume that a developer might fix any bug report if it is likely within his/her expertise. Thus, whether developers have obvious preferences on components or not will influence the effectiveness of *supervised* bug triage approaches.

Approach: Using the same dataset in **RQ1**, we look at whether the fixed bugs of each developer distributed equally on each component in a product by simply counting components of fixed bugs of a developer. If not, that means developers do have preferences on components in a project when they fix bugs.

Results: Table 3 shows the distributions of bug-fix activities of the eight developers listed in Table 1. As we can see, for all developers, most of their bug-fix activities are only in one component, e.g., developer Olivier_Thomann has fixed 1242 bugs in Eclipse JDT, among which 1227 bugs belong to component JDT Core.

We further study the average number of components that a developer has worked on in Eclipse JDT and Platform, we refer to this number as *ACnumber*. Results show that in JDT from Jan. 2002 to Jan. 2009, 97 developers had worked on it and the *ACnumber* is 1.44; in Platform there were 238 developers and the *ACnumber* is 1.81. This outcome means that usually a developer works on no more than two components of a project. Moreover, about 95% of most developers' bug-fix activities are in preferred components.

Most developers work on only one or two components in a product.

Results of our pilot studies motivate us to look deeper in practicing automatic bug triage, and inspire us to propose an *unsupervised* bug triage approach which can achieve higher *accuracy* without hurting the *diversity* of developer recommendation.

3. METHODOLOGY

3.1 Overview

Figure 3 shows the overview of our proposed bug triage approach. In FixerCache, each component of a product has a dynamic *developer cache* which contains prioritized developers according to developers' *activeness scores*. *Developer cache* is dynamically updated after each verification and resolution of bug report. In FixerCache, bug triage is modeled as follows: given an incoming bug report, we find the developer(s) with high fixing probability using *developer cache* of the component of this bug report. That is, if a developer has a higher *activeness score* in a component, he/she is supposed to take part in handling incoming bugs in the component.

Different from existing *supervised* approaches, FixerCache does not need time and space cost for filtering developers and extracting text-information from historical bugs for learning developers' bug-fix preferences. It does not need complex computation for recommending candidate developers, either. Thus, FixerCache possesses the following benefits.

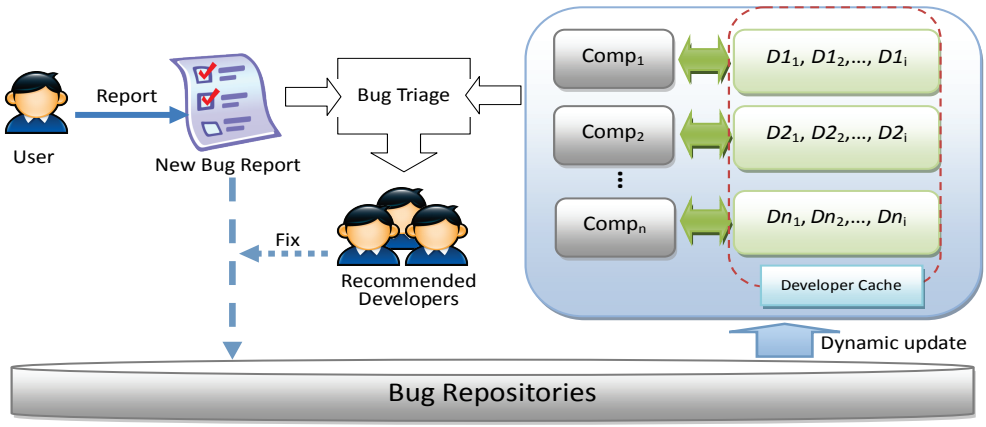


Figure 3. An overview of proposed FixerCache

1. Easy to implement, FixerCache does not need any complex computation. And it can be implemented simply by an instance of class "*java.util.Map*", which caches developers' *activeness scores* in a component. Obviously, FixerCache is more practical and efficient as well as time and space saving. Algorithm 1 illustrates how to implement proposed FixerCache.

2. FixerCache achieves higher prediction *accuracy* especially with top-10 recommendation list, which is up to 96.32%.

3. FixerCache achieves higher *diversity*, it does not filter any less active developers, and appropriate developers would have opportunities to fix bugs using FixerCache.

Algorithm 1. FixerCache
Input: a period of developers' fixing records ***H***, a new bug report ***R***
Output: list ***L*** (recommended developers)
1. CacheInitial(H);
2. DeveloperRecommendation(H, R, R.component);
3. Procedure CacheInitial(H)
4. DeveloperCache<component, develoeprCacheList>;// *instance of Map*
5. **for** entity *d* in *H* do
6. calcualteActivenessScore;
7. insert *d* to DeveloperCache;
8. **end for**;
9. Procedure DeveloperRecommendation(DeveloperCache, *R*, *R*.component)
10. CandidateList = NULL;
11. **for** entity *e* in DeveloperCache do
12. **if**(*e*.component == *R*.component)
13. CandidateList = *e*. develoeprCacheList.getTopNDevelopers;
14. **end if;**
15. **end for;**
16. **return** CandidateList;

The main data structure of FixerCache

3.2 Terminologies

Before we illustrate how FixerCache works, we introduce two important definitions used in this work.

DEFINITION1: *Cache Period. In order to recommend developers for fixing bugs in a component, FixerCache needs to cache developers' component-level fixing activities in the last N days, we refer to this period as Cache Period.*

Cache Period varies for different projects, because developers' behavior patterns [28], team structures [31], bug-fix patterns [14], and experiences [29] are different in different projects. It's hasty to say the longer *Cache Period* the better, developers may leave a community or move to other projects of the same community, which might cause extra space and time to maintain these developers' activities in the *developer cache* of a component and affect the performance of developer recommendation. We further study how to set *Cache Period* in Section4.2.

DEFINITION2: *Activeness Score. Given a developer d, the activeness score on component comp* $Score_{comp}(d)$ *is calculated based on the number of fixed bugs of d on component comp, time span between the time of the last fixing activity of d and the time of the latest fixed bugs add to component comp*:

$$Score_{comp}(d) = FixNum_{Cperiod}(d, comp) \Big/ e^{\left(t_2 - t_1 / 30\right)} \quad (3)$$

In the formula, $FixNum_{Cperiod}(d, comp)$ denotes the number of bugs developer *d* fixed in *Cache Period,* which is denoted as *Cperiod* (*Cperiod* >=0) in equation 3, t_1 is the timestamp of the last bug-fix activity of developer *d*, t_2 is the timestamp of the latest fixed bugs added to component *comp*, and both t_1 and t_2 are calculated by day. Obviously, the *activeness score* of a developer decays along with time if he/she does not fix bugs.

Table 4. Datasets of four projects in Eclipse and Mozilla (from 2002/01/01 to 2009/01/01)

Project	#Component	#Developer	#Bug
Eclipse JDT	6	97	32777
Eclipse Platform	20	238	17937
Mozilla Firefox	26	466	69195
Mozilla SeaMonkey	31	428	51038

3.3 Developer Cache and Its update

In FixerCache, after each fixed bug added to a component, the *activeness scores* of all developers will be recalculated in the component, and then *developer cache* of the component will be updated according to developers' new *activeness scores*.

3.4 Developer Recommendation

FixerCache recommends the most capable developers for a given incoming bug report *b* by the following two simple steps: First, it identifies the component of *b* from the attribute "component" of bug report *b*. Then, FixerCache recommends the top-N developers in the *developer cache* of component of bug report *b*.

4. EMPIRICAL EVALUATION

We evaluated FixerCache on four products of two large open source projects, i.e., Eclipse and Mozilla, which are widely used in bug triage methods [1-6,8-11,16,28,29]. The datasets are listed in Table 4. We evaluated FixerCache with various parameters for *Cache Period* and compared it with two machine learning-based bug triage approaches, i.e., SVM and NB, which adopted as benchmarks in almost all existing bug triage approaches.

Note that we did not compare FixerCache with other bug triage approaches which convinced could achieve high *accuracy* (bigger than 80% on average) in [5,6,9,10] or our previous work [29], because all these approaches need a refined selection of developers (usually only select 10% developers as their experiment datasets), as presented in our pilot study (see Section 2.3), their *diversity* values are up to 40%, while in FixerCache we do not filter any developer. Thus, we argue that it is not appropriate to compare FixerCache with these approaches. We ran all the experiments on a PC with a 2.8GHz CPU and an 8GB RAM.

4.1 Evaluation Measures

We use two metrics to evaluate the performance of bug triage approaches, i.e., *accuracy* and *diversity*.

Accuracy: We evaluate the results of bug triage with the *accuracy* of top 1, top 5 and top 10 developer recommendation. The *accuracy* is defined as $Accuracy = \frac{\#correctly\ predicted\ bug\ reports}{\#all\ the\ bug\ reports}$ based on the recommendation list.

Diversity: We propose to use this metric to measure the capacity of modelling real bug triage practice of a bug triage approach.

And *diversity* is calculated by the ratio of recommended developers in all the unique developers who fixed bugs in a project (see Section 2.3).

4.2 Cache Period Setting

FixerCache allows to cache developers' activities in a period. For setting an appropriate *Cache Period*, we tuned different values for *Cache Period* and explored their influence on performance of FixerCache via prediction *accuracy* and *diversity*. Using datasets listed in Table 4, we ran FixerCache with various values of *Cache Period*, from 0.5 months to 6 months (increased by 0.5 months). For each value, we examine the performance of predicting fixers of bugs in the next 1 month by measuring both the average *accuracy* and *diversity*.

We present the results of top-1, top-5 and top-10 prediction *accuracy* and *diversity* with different values of *Cache Period* in Figures 4 to 6. As seen, the performance of FixerCache on four projects varies with different values of *Cache Period*, and the prediction *accuracy* and *diversity* peak at some values of *Cache Period*. This implies that setting a suitable value of *Cache Period* is important for better practicing FixerCache. For example, in Eclipse Platform, when *Cache Period* is equal to 1 month, top-10 *accuracy* of FixerCache is 96.32% and *diversity* is 76.07%. However, top-10 *accuracy* is 89.02% and top-10 *diversity* is only 57.88% when *Cache Period* is equal to 6 months.

We suggest that for better practicing FixerCache, before triaging bugs, setting a suitable *Cache Period* value should be done. From empirical studies on the four projects used in our experiments, we find that usually a suitable value for *Cache Period* ranges from 1 month to 3 months. Thus, in order to practice FixerCache efficiently, a developer group/software community should have at least 1 month's bug fixing records for caching developers' activeness.

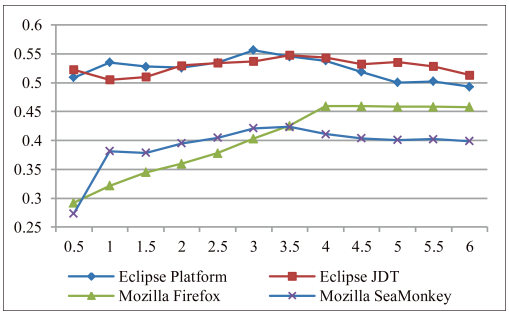
4.3 Performance Comparison

For comparison, we used Weka [7] to implement SVM and NB based bug triage approaches in [1], which are widely adopted as benchmarks in bug triage approaches [2,6,8-11,16,28,29]. In order to make these two approaches comparable with FixerCache, we did not filter any developers in the datasets.

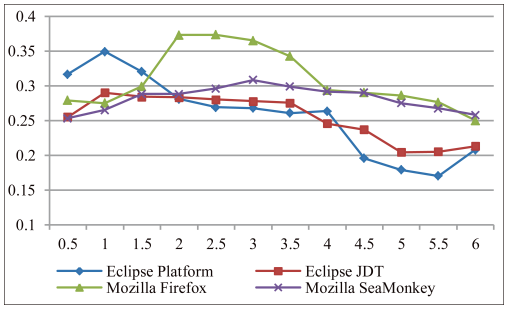
As described in our pilot study (see, Section 2.3), we employ incremental learning to evaluate the result of bug triage, we sort bug reports in chronological order and divide them into 11 folds, and execute 10 rounds to investigate accuracies of all the folds. In each round, we calculate both prediction *accuracy* and *diversity* with top-1, top-5 and top-10 recommendation list. The average *accuracy* and *diversity* of SVM and NB and their comparison with FixerCache are shown in Table 5. For the *Cache Period* we use the recommended values in Section 4.2.

As seen, FixerCache outperforms SVM and NB based bug triage approaches both in *accuracy* and *diversity*, especially with top-10 recommendation list. FixerCache improves the SVM and NB by about 20% in *accuracy* on average, and 15% in *diversity* on average.

We further explore the time cost of SVM and NB based bug triage approaches and FixerCache. We measure the time of data processing, model building and training and developer recommending of these bug triage approaches. Results are shown in Table 6, for SVM and NB, they need extra time for some natural language processes, i.e., removing stop words, stemming and extracting string vectors from the summaries and descriptions of bug reports. Since FixerCache does not need these processes, there is no time cost on data processing. As for developer recommending, FixerCache does not need complicated computation, results in Table 6 show that it reduces bug triage time from hours of *supervised* approaches to seconds.

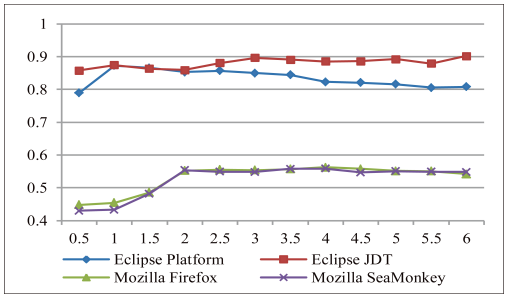


(a) Top-1 developer recommendation *accuracy*

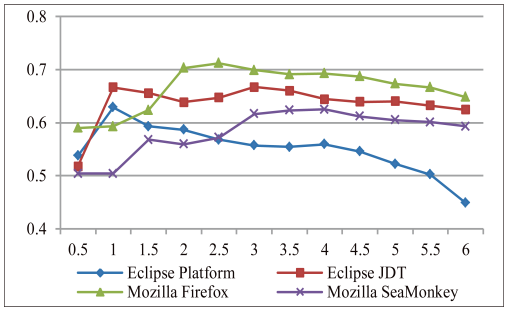


(b) Top-1 developer recommendation *diversity*

Figure 4. Performance of top-1 developer recommendation with various *Cache Period* values



(a) Top-5 developer recommendation *accuracy*



(b) Top-5 developer recommendation *diversity*

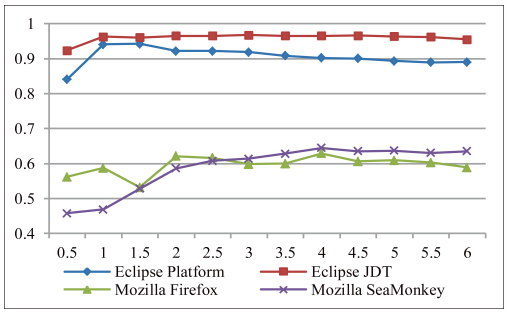
Figure 5. Performance of top-5 developer recommendation with various *Cache Period* values

Table 5. Top-1, Top-5 and Top-10 Prediction *Accuracy* (%) and *Diversity* (%)

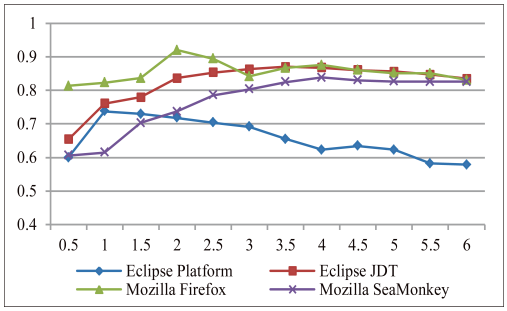
Project	Metric	Top1			Top5			Top10		
		SVM	NB	FixerCache	SVM	NB	FixerCache	SVM	NB	FixerCache
Eclipse JDT	*Accuracy*	23.92	23.95	54.32	55.12	54.61	88.56	69.41	69.68	96.32
	Diversity	30.00	30.67	28.91	57.14	66.19	66.73	59.52	70.95	86.02
Eclipse Platform	*Accuracy*	21.63	22.73	53.78	43.58	47.67	84.45	55.95	57.38	93.02
	Diversity	30.48	42.76	34.92	54.26	60.21	62.91	60.64	72.34	73.72
Mozilla Firefox	*Accuracy*	18.62	16.67	45.92	37.76	46.09	55.56	51.85	57.41	61.65
	Diversity	8.20	30.07	30.35	16.39	42.62	61.21	22.95	54.92	84.16
Mozilla SeaMonkey	*Accuracy*	12.62	15.02	44.38	40.10	36.58	54.63	47.61	47.28	62.50
	Diversity	18.18	29.85	37.89	27.27	40.90	71.66	30.30	46.96	91.67

Table 6. Comparison of Processing Time (s: seconds, m: minutes, h: hours)

Project	Data processing			Model building and training			Developer recommending		
	SVM	NB	FixerCache	SVM	NB	FixerCache	SVM	NB	FixerCache
Eclipse JDT	30m		N/A	3h	4.5h	0.1s	3m	4m	0.05s
Eclipse Platform	25m		N/A	2.7h	3.8h	0.09s	3m	3.8m	0.03s
Mozilla Firefox	37m		N/A	3.5h	5h	0.13s	3.2m	4.3m	0.07s
Mozilla SeaMonkey	33m		N/A	3.4h	4.6h	0.12s	3.2m	4.2m	0.1s



(a) Top-10 developer recommendation *accuracy*



(b) Top-10 developer recommendation *diversity*

Figure 6. Performance of top-10 developer recommendation with various *Cache Period* values

5. DISCUSSION

In FixerCache, the *developer cache* of a component is dynamically updated after each verification and resolution of a bug report in this component. In theory, the less time span between the last update of *developer cache* and an incoming bug, the higher discriminative power FixerCache achieves. However in real practice of bug resolution, fixing bugs usually take days even months. Thus, time span between the last update of *developer cache* and an incoming bug varies. We refer to this time span as *Prediction Range* shown in Figure 7.

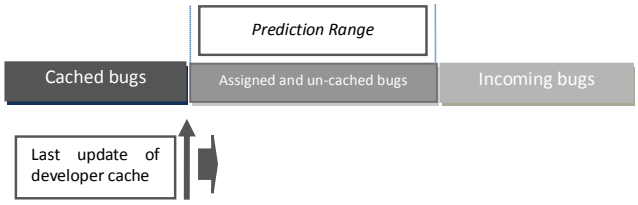


Figure 7. *Prediction Range* in FixerCache

We simply divide bug reports into three different catalogs based on their statuses. We refer to bugs which are resolved and cached as "Cached bugs", bugs which are in process of resolution as "Assigned and un-cached bugs", bugs which are just submitted to bug repositories and have not been handled as "Incoming bugs". The *developer cache* of a component will be dynamically updated after the resolution of each bug.

For better understanding the influence of *Prediction Range* on the performance of FixerCache, using datasets shown in Table 4, we examine the efficiency of FixerCache via average prediction *accuracy* and *diversity* with different size of *Prediction Range*. Note that, we only examine the performance with top-10 recommendation list, since the experiments in Section 4.3 show FixerCache can achieve impressive high *accuracy* and *diversity* with top-10 recommendation list. In our experiments, the size of *Prediction Range* varies from 0.5 months to 6 months and increases by 0.5 months. We use recommended values in Section 4.2 to set *Cache Period* for the four projects used in our experiments.

Figures 8 and 9 show the results. As seen, prediction *accuracy* and *diversity* of four projects have the same behavior. With the increasing of *Prediction Range*, both *accuracy* and *diversity* decrease. This outcome is especially obvious in Mozilla Firefox. When the *Prediction Range* increases from 0.5 months to 6 months, its *diversity* decreases from 83.90% to 23.45%, one possible reason for this fact is that developer teams evolved quickly in Mozilla [31], many new developers came and left, this fact may lower the performance of FixerCache.

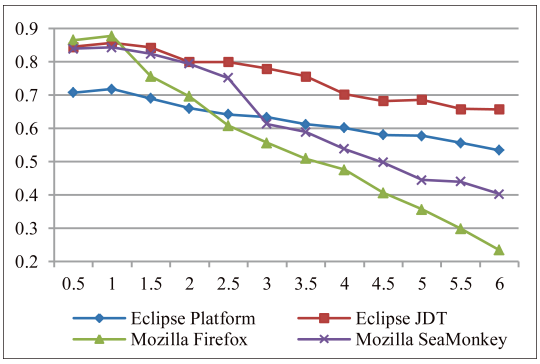


Figure 8. *Diversity* of FixerCache with different *Prediction Range*

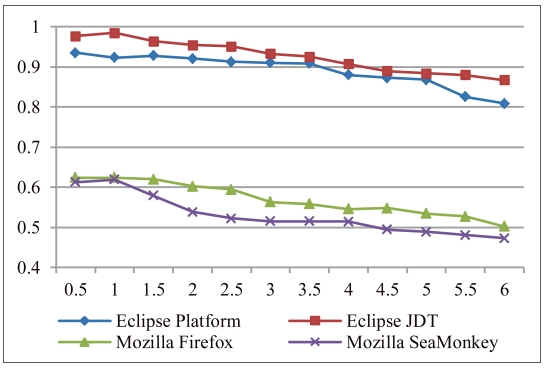


Figure 9. *Accuracy* of FixerCache with different *Prediction Range*

Tendencies of prediction *accuracy* and *diversity* in Figures 8 and 9 show that when *Prediction Range* ranges from 0.5 months to 2.5 months the decreasing of both *accuracy* and *diversity* in four projects is slight. This result suggests that the *Prediction Range* should not be bigger than 2.5 months for efficient practicing FixerCache.

6. THREATS TO VALIDITY

In this section we present some of the threats to the validity of this study.

6.1 External Validity

In this work, we investigate the performance of proposed FixerCache on four products of two large open source software projects, i.e., Eclipse and Mozilla. However, it is possible that our approach may not work well on some closed-source software (e.g., commercial software) or small scale open source software projects, where developers' behaviors may be different with that of Eclipse and Mozilla. Whether our proposed approach is feasible for these software projects should be further investigated.

6.2 Internal Validity

In this paper, we assume that the *activeness score* of a developer has an exponential relationship with the number of his/her already fixed bugs, the interval between the timestamp of his/her last fixing activities and the timestamp of the latest fixed bug added to related component. This is hard to validity. In the future, we plan to further study the influence of different *activeness score* functions, e.g., social influence-based method in [32], dynamic activeness model in [33], on the efficiency of FixerCache.

For different projects, *Cache Period* is a significant factor that affects the performance of FixerCache, e.g., in Eclipse the best value of *Cache Period* is 1 month, while in Mozilla the value is 2-3.5 months. Thus, further questions may be raised, e.g., what causes the different suitable *Cache Period* among projects? How to set an appropriate *Cache Period* for different products? Further studies should be done for answering these questions.

7. RELATED WORK

There exists machine learning (ML), information retrieval (IR) and hybrid (e.g., social network analysis + ML/IR) *supervised* bug triage approaches leverage text-information-based expertise of developers for semi-automatic bug triage.

ML based approaches: Čubranić and Murphy [2] first modeled bug triage as a text classification problem. They extracted the title, description, and keywords from bug reports to build a Naïve Bayes classifier to recommend developers for bug fixing. In their work, no developers were filtered and the prediction *accuracy* was around 30% on Eclipse bug repositories from Jan to Sep-2002. Anvik et al. [1] improved the above work with filtering out unfixed bug reports and less active developers (developers who fixed less than 50 bugs). They used three different ML classifiers, i.e., SVM, Naïve Bayes, and C4.5. And the *accuracy* of bug triage was increased up to 64%. Lin et al. [13] explored bug triage problem with proprietary software project in a Chinese software company using ML with SVM and C4.5 classifiers on both Chinese text and the other non-text information (e.g., bug type, bug class, phase ID, submitter, model id, bug priority). Their experiments involved 2,576 bug reports, and achieved the prediction *accuracy* of up to 77.64% (ignoring model ID) and 63% (considering model ID).

IR based approaches: Matter et al. [16] used Vector Space Model (VSM) to model a developer's expertise using a vector of frequent terms extracted from their source code contributions. For bug triage, they compared the vector of a new bug with vectors of developers' expertise. They used eight years of Eclipse development bug data as a case study including 130,769 bug reports. They achieved the prediction *accuracy* up to 71.0% with top-10 recommendation list. Tamrawi et al. [6] leveraged fuzzy sets (vectors of selected terms extracted from fixed bugs of developers) to represent the capable developers of fixing bugs related to a technique issue. They compared the vector of terms of a new bug with fuzzy sets of technique aspects to recommend developers. They evaluated their methods with 10%-40% active developers (filtered less active developers) on 6 projects, results showed that their method achieved the *accuracy* up to 83% with top-5 recommendation list. Shokripour et al. [8] proposed a location-based approach for bug triage based on noun terms extracted from source codes. Their approach achieved an *accuracy* of 89.41% on well filtered Eclipse data set (only contains 9 active developers) and 59.76% on Mozilla data set (only contains 57 active developers). Xuan et al. [19] proposed a semi-supervised text classification approach for bug triage combined Naïve Bayes classifier and expectation-maximization, and their approach could achieve the *accuracy* up to 48%. Xie et al. [10] used topic models to represent developers' interest and

expertise on bug resolving activities based on their historical bug resolving records. Experimental results shown their method can achieve recall up to 82% on Eclipse dataset with top-5 recommendation list and 50% on Mozilla Firefox with top-7 recommendation list. Xia et al. [30] improved the above work by combining ML-KNN and topic model.

Hybrid approaches: Jeong et al. [4] first introduced the idea of bug tossing graphs based on Markov chains. Their Markov-based model derived from the patterns of bug tossing during the fixing process of bugs. Based on the tossing graphs they re-ranked recommendation results of machine learning classifiers, i.e., Naïve Bayes and Bayesian Network. Results of experiments on Eclipse and Mozilla datasets shown that their ML+tossing graphs methods achieved the *accuracy* up to 72% with top-5 recommendation list. Bhattacharya and Neamtiu [5] improved the above work with refined classification using additional attributes, intra-fold updates during training and well selected datasets (developers who fixed more than 50 bugs), Their techniques can achieve up to 83.62% prediction *accuracy* in bug triaging with top-5 recommendation list. Xuan et al. [3] leveraged social network analysis techniques to prioritize developers, and they used priority of developers to re-rank recommendation results of machine learning classifiers, i.e., Naïve Bayes and SVM. Their approach achieved the *accuracy* up to 67.61% with top-5 recommendation list on Eclipse and Mozilla data. Naguib et al. [9] proposed a bug triage method leveraged developers' activities in bug tracking repository and LDA-SVM-based developers' expertise characterized by text-information to triage bugs. Their method achieved an average *accuracy* of 88% with top-10 recommendation list on well selected 753 bug reports from three open source projects. Our previous work [28,29] explored developers' collaboration and contribution with heterogeneous network analysis of bug repositories. Then we leveraged such collaboration and contribution to improve bug triage by re-ranking recommendation results of machine learning classifiers, i.e., Naïve Bayes and SVM. Results of experiments on Eclipse and Mozilla data (we filtered developers who had fixed less than 50 bugs) shown that our methods could achieve an *accuracy* up to 89.39% with top-5 recommendation list.

Compared with the above bug triage approaches, FixerCache has three obvious advantages. First, FixerCache achieves higher *accuracy* especially with top-10 recommendation list which achieves up to 96.32%. Second, FixerCache achieves higher *diversity*, since it does not filter any less active developers. It's a win-win solution in balancing the *diversity* and *accuracy* of developer recommendation in bug triage. Third, FixerCache spends less time and space in data collecting, model building and training, and recommending developers, since FixerCache only maintains a dynamic developer list according to developers' *activeness scores* for each component, this also makes it more practical and efficient as well as time and space saving.

8. CONCLUSION

In this paper, we empirically address three limitations of *supervised* bug triage approaches. The first one is that the increasing similarities of text-information-based developers' expertise lower the discriminative power of *supervised* bug triage approaches. The second one is that filtering developers with less bug-fix activities lowers the *diversity* of developer recommendation. The third one is that triaging bugs primarily based on expertise characterized by text-information does not take developers' activities scope in the project into account. To address the three limitations of text-information-based *supervised* bug triage approaches, we propose FixerCache, an *unsupervised* bug triage approach based on developers' *activeness scores* in components of products.

For better evaluating bug triage approaches, we propose to use not only *accuracy* but also *diversity* to measure their performances. Evaluation on four projects of Eclipse and Mozilla datasets shows that FixerCache achieves higher *accuracy* and *diversity* than existing bug triage approaches. Moreover, since FixerCache does not need to learn developers' text-information-based bug-fix preferences, it could reduce bug triage time from hours of *supervised* approaches to seconds.

The datasets and codes of our work are available here: http://itechs.iscas.ac.cn/cn/material/wangsong/fixercache.zip.

9. ACKNOWLEDGMENTS

This research was supported in part by National Natural Science Foundation of China under Grant Nos. 91218302, 61073044, 91318301, 71101138, and 61303163; National Science and Technology Major Project under Grant Nos. 2012ZX01039-004; Beijing Natural Science Fund under Grant No.4122087; State 863 High Technology R & D Project No. 2012AA011206.

10. REFERENCES

[1] J. Anvik, L. Hiew, and G.C. Murphy, "Who Should Fix This Bug?," Proc. 28th Intl. Conf. Software Engineering (ICSE '06), May 2006, pp. 361-370.

[2] D. Čubranić and G.C. Murphy, "Automatic Bug Triage Using Text Categorization," Proc. 16th Intl. Conf. Software Engineering & Knowledge Engineering (SEKE '04), Jun. 2004, pp. 92-97.

[3] J. Xuan, H. Jiang, Z. Ren, and W. Zou, "Developer Prioritization in Bug Repositories," Proc. 34st Intl. Conf. Software Engineering (ICSE '12), June.2012, pp. 25-35.

[4] G. Jeong, S. Kim, and T. Zimmermann, "Improving Bug Triage with Tossing Graphs," Proc. 17th ACM SIGSOFT Symp. Foundations of Software Engineering (FSE'09), Aug. 2009, pp. 111-120.

[5] P. Bhattacharya and I. Neamtiu, "Fine-Grained Incremental Learning and Multi-Feature Tossing Graphs to Improve Bug Triaging," Proc. 26th IEEE Intl. Conf. Software Maintenance (ICSM '10), Sept. 2010, pp. 1-10.

[6] A. Tamrawi, T.T. Nguyen, J.M. Al-Kofahi, and T.N. Nguyen, "Fuzzy Set and Cache-Based Approach for Bug Triaging," Proc. 19th ACM SIGSOFT Symp. Foundations of Software Engineering (FSE '11), Sept. 2011, pp. 365-375.

[7] I.H. Witten, E. Frank, and M.A. Hall, Data Mining: Practical Machine Learning Tools and Techniques, 3rd ed. Morgan Kaufmann, Burlington, MA, 2011.

[8] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani, "Why So Complicated? Simple Term Filtering and Weighting for Location-Based Bug Report Assignment Recommendation," Proc. 10th IEEE Working Con. on

Mining Software Repositories (MSR' 13), May. 2013, pp.2-11.

[9] H. Naguib, N. Narayan, B. Brugge, and D. Helal, "Bug Report Assignment Recommendation using Activity Profiles," Proc. 10th IEEE Working Con. on Mining Software Repositories(MSR'13), May. 2013, pp.22-30.

[10] X. Xie, W. Zhang, Y. Yang, and Q. Wang, "Dretom: developer recommendation based on topic models for bug resoluton," Proc. 8th International Conferenece on Predictive Models in Software Engineering (PROMISE'12), Mar. 2012, pp. 19-28.

[11] W. Wu, W. Zhang, Y. Yang, and Q. Wang, "Drex: Developer recommendation with k-nearest-neighbor search and expertise ranking," in 18th Asia Pacific Software Engineering Conference (APSEC'11), Dec. 2011, pp. 389-396.

[12] W. Zhang, Y. Yang, and Q. Wang, "An empirical study on identifying core developers using network analysis," Proc. 2nd Intl. Workshop on Evidential Assessment of Software Technologies (EAST'12), Sep. 2012, pp. 43-48.

[13] Z. Lin, F. Shu, Y. Yang, C. Hu, and Q. Wang. "An empirical study on bug assignment automation using Chinese bug data," Proc. 3th ACM/IEEE Intl. Symp. Empirical Software Engineering and Measurement (ESEM'09), Oct. 2009, pp 451-455.

[14] E. Murphy, T. Zimmermann, C. Bird, and N. Nagappan, "The design of bug fixes," Proc. 35th Intl. Conf. Software Engineering (ICSE '13) May. 2013, pp.332-341.

[15] A. Gunawardana and G. Shani, "A Survey of Accuracy Evaluation Metrics of Recommendation Tasks," Journal of Machine Learning Research, Vol. 10, Dec. 2009, pp.2935-2962.

[16] D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning Bug Reports using a Vocabulary-based Expertise Model of Developers," Proc. 6th IEEE Working Con. on Mining Software Repositories (MSR' 09), May. 2009, pp 131-140.

[17] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Permraj, and T. Zimmermann, "What Makes a Good Bug Report," Proc. 16th ACM SIGSOFT Symp. Foundations of Software Engineering (FSE'08), Nov. 2008, pp. 308-318.

[18] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," Proc. 35th Intl. Conf. Software Engineering (ICSE'13), May. 2013, pp.392-401.

[19] J. Xuan, H. Jiang, Z. Ren, J. Yan, and Z. Lou, "Automatic Bug Triage Using Semi-Supervised Text Classification," Proc. 22th. Intl. Conf. Software Engineering & Knowledge Engineering (SEKE' 10), Jul. 2010, pp.209-214.

[20] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Permraj, and T. Zimmermann, "Quality of bug reports in Eclipse," Proc. OOPSLA workshop on eclipse technology eXchange (eclipse'07), Oct. 2007, pp. 21-25.

[21] J. Xie, M. Zhou, and A. Mockus, "Impact of Triage: A Study of Mozilla and Gnome," Proc. 7th ACM / IEEE Intl. Symp. Empirical Software Engineering and Measurement (ESEM'13), Oct. 2013, pp. 247-250.

[22] "The gnome bugsquad," https://live.gnome.org/Bugsquad, 2012.

[23] "Mozilla triage guide – harnessing the flood of community," https://wiki.mozilla.org/QA/Triage, 2010.

[24] J. Han and M. Kamber, Data Mining: Concepts and Techniques. Morgan Kaufmann Publishers, USA

[25] J. R. Sean M. McNee and J. A. Konstan, "Accurate is not always good: How accuracy metrics have hurt recommender systems," In extended abstracts on Human factors in computing systems (CHI'06), 2006, pp 1097-1101.

[26] M. Zhang and N. Hurley, "Avoiding monotony: Improving the diversity of recommendation lists," Proc. 2nd ACM International Conf. Recommender Systems (RecSys'08), Oct.2008, pp. 123-130.

[27] C. Yu, L. Lakshmanan, and S. A. Yahia, "It takes variety to make a world: diversification in recommender systems," Proc. 12th Intl. Conf. Extending Dtabase Technology: Advances in Database Technology (EDBT'09), Mar. 2009, pp.368-378.

[28] S. Wang, W. Zhang, Y. Yang, and Q. Wang, "DevNet: Exploring Developer Collaboration in Heterogeneous Network of Bug Repositories," Proc. 7th ACM / IEEE Intl. Symp. Empirical Software Engineering and Measurement (ESEM'13), Oct. 2013, pp 193-202.

[29] W. Zhang, S. Wang, Y. Yang, and Q. Wang "Heterogeneous Network Analysis of Developer Contribution in Bug Repositories," International Conference on Cloud and Service Computing (CSC'13), Nov. 2013, pp 98-105.

[30] X. Xia, D. Lo, X. Wang, and B. Zhou "Accurate developer recommendation for bug resolution," 20th Working Conf. Reverse Engineering (WCRE'13), Oct. 2013, pp.72-81.

[31] Q. Hong, S. Kim, S.C. Cheung, and C. Bird, "Understanding a Developer Social Network and its Evolution," Proc. 27th IEEE Intl. Conf. Software Maintenance (ICSM '11), Sept. 2011, pp. 323-332.

[32] Z. Wen and C. Y. Lin, "On the Quality of Inferring Interests From Social Neighbors," Proc. 16th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD'10), Jul. 2010, pp. 373-382.

[33] S. Lin, X. Kong, and P. S. Yu, "Predicting trends in social networks via dynamic activeness model," Proc. 22nd ACM Intl. Conf. on Information & Knowledge Management (CIKM'13), Oct. 2013, pp. 1661-1666.

[34] M. Ge, C. D. Battenfeld, and D. Jannach, "Beyond accuracy: evaluating recommender systems by coverage and serendipity" Proc. 4nd ACM Intl. Conf. Recommender Systems (RecSys'10), Sep.2010, pp. 257-260.

[35] J. Xu, Y. Gao, S. Christley, and G. Madey, "A Topological Analysis of the Open Source Software Development Community," Proc. 38th Annual Hawaii Intl. Conf. on System Sciences (HICSS'05), Jan. 2005, Vol. 7.