Developer Recommendation with Awareness of Accuracy and Cost

*Jin Liu, Yiqiuzi Tian
State Key Lab of Software Engineering
Computer School, Wuhan University
Wuhan, China
*Corresponding author
jinliu@whu.edu.cn
tianyiqiuzi@whu.edu.cn

Liang Hong
School of Information Management
Wuhan University
Wuhan, China
hong@whu.edu.cn

Zhou Xu
State Key Lab of Software Engineering
Computer School, Wuhan University,
Wuhan, China
zhouxullx@whu.edu.cn

***Abstract*—As the scale and complexity of software products increase, software maintenance on bug resolution has become a challenging work. In the process of software implementation, developers often use bug reports, source code and change history to help solve bugs. However, hundreds of bug reports are being submitted every day. It is time-consuming and effortless for developers to review all the bug reports. To facilitate the assignment of bug reports, existing developer recommendation systems typically recommend the developer who has the fullest potential. However, bug reports are highly varied; time that the developers may spend fixing them is also important. To address the problem of developer recommendation, we propose a developer recommendation system with awareness of accuracy and cost (DRAC). This recommendation system is based on modern portfolio theory by striking a balance between accuracy and cost (time). We evaluate our approach with experiments on data collected from Bugzilla[1].**

Keywords—Recommendation System; Portfolio Theory; Bug Triage; Accuracy and Cost

I. INTRODUCTION

Software development life cycle usually consists of six phases, i.e. requirements analysis, software design, implementation, testing, integration and maintenance. In the phase of implementation, many bugs appear with causes stemming from various stages. Previous studies have shown that more than 45% of modern software development is used to locate and fix bugs [1][2]. Generally, developers upload bug reports to bug-tracking platform, such as *Bugzilla*[1], *Mantis*[2], *Trac*[3] and *Redmine*[4] with a fixed format when they encounter bugs during implementation phase. It can also be considered as a crowdsourcing problem[3][4]. The bug reports are viewed and solved by the users of the platform, and it can be very time-consuming. Having a framework that makes predictions about the right developer with higher accuracy and less time is essential to shorten the lifecycle of the bugs [5]. Another fact about the bug report is that its volume is big and grows at a high rate at the same time. So strategy should be taken to response quickly when a new bug report is brought up [6].

Above all, the main problem in this area is the absent consideration of time cost. Existing works only do study on optimizing the possibility of finding the developer with the most similar history bug reports, but they fail to explore the cost. However, cost is very important, for it implies the time needed to get this bug fixed by a specific developer, and time cost has a great influence on software development process.

These observations lead to two goals: First, we need to formulate a recommendation framework to make predictions about developers with the consideration of both accuracy and cost. Second, to make quick response when a new bug report is brought up, we need to come up with a strategy managing the data efficiently.

In this paper, we propose a developer recommendation system with awareness of accuracy and cost (DRAC), which is modeled on the description of the bug reports. We divide the whole process into two phases - the offline learning stage and online recommendation stage. The first stage aims at learning the value of parameters; and the second stage uses the portfolio theory to make prediction about the best developer with a balance between accuracy and time cost. Finally, we evaluate our developer recommendation approach with experiments on a large-scale real-world dataset collected from project "Eclipse" on Bugzilla [7]. The experiment results validate the effectiveness and efficiency of our approach.

II. BACKGROUND

A. Data with Big Data Features

Big data has drawn huge attention from researchers recently [8]. Gartner listed big data as the second place in "Top 10 Critical Tech Trends for the Next Five Years" [9]. Technically speaking, big data is a collection of very huge dataset with a great diversity of types, so it becomes difficult to process by using state-of-the-art data processing approaches or traditional data processing platform. Big data has three features: variety, volume and velocity. Bug reports repository complies with these three specifications with its different-formatted components, large data scale and high growth rate. In Fig. 1, we can see that the size of bug report repository increases at a steady speed. At the end of the year 2015, the total amount of bug reports has passed 480,000, with nearly 50 reports brought up every day. It's essential to reduce the volume of data before making recommendation. The applicable way includes feature selection and instance selection [10].

[1] http://www.bugzilla.org/.
[2] http://www.mantisbt.org/.
[3] http://trac.edgewall.org/.
[4] http://www.redmine.org/.

This work is partly supported by the grants of National Natural Science Foundation of China (61572374, U1135005, 61303025).
DOI reference number: 10.18293/SEKE2016-125

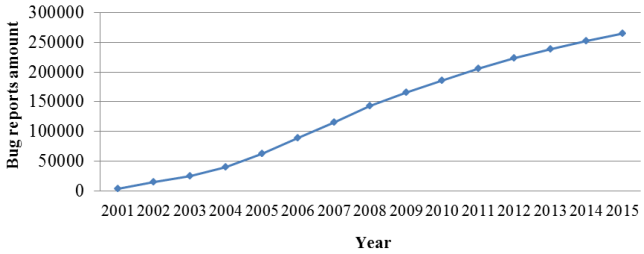

Figure 1. The increasing size of bug report repository

B. Recommendation System

Recommendation systems have become extremely common in recent years and are applied in a variety of applications. They typically produce a list of recommendations. Main methods in the area include content-based recommendation (CBR), collaborative filtering (CF) and association rule-based recommendation (ARBR). Each method adapts to different scenarios [11].

Considering our problem, the data source has two different forms [12]. The first is called metadata. It contains data like ID of the bug, the platform the bug belongs to, keywords and so on. The amount of metadata is fixed, and its value has a specific range. Using metadata to classify bug reports can be very hard, for that the information it implicates is very limited. The second is text data, which consist of description, summary and other developers' comments. With unlimited size and range, text data contain much richer information. So, different data sources should be weighted differently. In the field of developer recommendation, to accurately assign developers to bug reports, recent research treats it as a problem that optimizes recommendation accuracy and proposes solutions that are essentially instances of content-based recommendation (CBR). CBR mainly uses the content of the item and make predictions based on the similarity between contents. In this case, the data we use are the text content of bug reports.

In the recommendation system about bug reports, related researches aim at recommending right developers to bug reports which only consider the recommendation accuracy with old bug reports and the source code as data source. The main technology includes machine learning [13][14], information retrieval technique [14][15], tossing graph[16], fuzzy set[17] and Euclidean distance [18][19]. Compared to DRAC, this recommendation strategy has an obvious flaw. It only recommends the right developer who is capable of solving the problem, but doesn't consider the time the developer may spend on solving it. For urgent bug reports which have high severity level, it can be devastating to developing process.

C. Modern Portfolio Theory

Modern portfolio theory is originally proposed in the field of finance, which focus on the investment problem of financial market [20]. It's a mathematical framework for assembling a portfolio of assets such that the expected return is maximized for a given level of risk. Its key insight is that an asset's risk and return should not be assessed by itself, but by its contribution to a portfolio's overall risk and return. For example, an investor often wants to select a portfolio of n stocks with a fixed investment budget, which will provide the maximum future return and the minimum risk. In our problem, the stocks can be regarded as bug reports, the future return and risk can be regarded as accuracy and cost of recommending bug reports.

In this problem, each bug report is taken as a recommender which recommends its own fixer to the considered bug report. When the modern portfolio theory is applied, risk vector and return risk are both needed, which in our case are cost vector and accuracy vector individually. By combining these two vectors, portfolio theory generates the weight of each bug report, i.e. the weight of the recommender.

III. Approach

In this section, we clarify the basic algorithm, framework of our recommendation system and the recommending strategy we used.

A. Similarity Computation

We need to get the similarity between bug reports in both online and offline stages. It's also used as input of the portfolio based algorithm in the last step. So in this section, we clarify the computational formula to get it.

Each of the bug reports in the repository is turned into a word count vector and a topic vector. The words in bug reports are collected in a dictionary, and each of the word is assigned with a unique label. Formally, w_a and t_a are in the form of $w_a(w_{a,1}, w_{a,2}, \ldots, w_{a,i}, \ldots, w_{a,n})$ and $t_a(t_{a,1}, t_{a,2}, \ldots, t_{a,i}, \ldots, t_{a,n})$, where a refers to the bug report a, $w_{a,i}$ means the times that the word i appears in the bug report a, and $t_{a,i}$ means the possibility that the bug report a belongs to the i^{th} topic.

With these two vectors, we calculate similarities between the new bug report and each bug report in the repository, which include *SimilarityW(a,u)* and *SimilarityT(a,u)*. They are the similarity between the word vector and topic vector of the bug report a,u respectively. The word vector similaritiy between bug report a and u is defined as

$$SimilarityW(a,u) = \frac{2 \times |I_a \cap I_u|}{|I_a| + |I_u|}, \quad (1)$$

where $|I_a \cap I_u|$ is the total count of words that appear in both the two bug reports, and $|I_a|$ and $|I_u|$ are the total number of words appearing in bug reports a and u, respectively. Both of the number of words that bug report a and u share and their corresponding counts have an influence on the similarity. When the co-appearing word count $|I_a \cap I_u|$ is small, the significance weight *SimilarityW(a,u)* will decrease the similarity estimation value between the bug reports a and u. Since the value *SimilarityW(a,u)* is between the interval of [0,1], the closer the value is to 1, the more similar bug reports a and u are. The similarity between the topic possibility vectors is defined as

$$SimilarityT(a,u)' = \sqrt{\sum (t_{a,i} - t_{u,i})^2}, \quad (2)$$

where i is the label of the topic, $t_{a,i}$ and $t_{u,i}$ refer to the possibility that bug report a and u belongs to topic i,

respectively. We take the cosine distance between them as their distance. The interval of *SimilarityT(a,u)* is [0,+∞]. To normalize the topic similarity, we take its reciprocal and add 1 to the denominator avoiding the similarity to be infinite. Thus, the topic similarity is

$$SimilarityT(a,u) = \frac{1}{SimilarityT(a,u)' + 1}. \quad (3)$$

We can see the range of similarity of topic *SimilarityT(a,u)* is [0,1], which is comparable with *SimilarityW(a,u)*.

B. Recommendation Framework

Here, we first define the problem of developer recommendations with similarity and cost awareness, and then clarify the recommendation framework.

DEFINITION 1 (PROBLEM STATEMENT). *Given a new bug report b, the goal of recommendation with accuracy and cost awareness is to build an optimal ranked list of developers based on both the developers' possibility of solving the problem and the time they need to solve it.*

The above problem statement raises two issues:

- How to get developers' possibilities of solving the bug report and produce a ranked list.
- How to combine the risk-based rank list with the time-based ranked list to produce final ranking to strike a balance between accuracy and cost, which means lessen the time without much influence to the accuracy.

There are often many developers as candidates for recommendations. Thus, how to efficiently manage developers for recommendation is also an open question. To that end, we propose a novel recommendation framework to solve these problems.

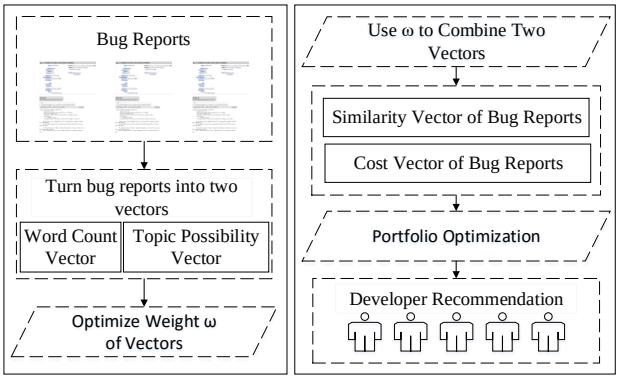

Figure 2. The recommendation framework

As we can see in Fig. 2, the whole recommendation framework contains two stages. The first stage is offline learning stage, where we learn the value of parameters that is needed in the second stage. And in online recommendation stage, we use modern portfolio theory [21] to combine the accuracy with the time cost to make recommendation about the best developer to solve the bug report. The specific method we apply will be clarified in the following specifications.

1) Offline Learning Stage

In this stage, we focus on determining the weight between the word count vector and the topic possibility vector.

These two vectors are different. Their contribution to the final prediction should differ too. So we need to determine their influences on the final prediction result.

For each bug report in testing set, we calculate their similarities and recommend the developer of the bug report which has the biggest similarity, and get the count of right recommendations. By varying the value of ω, we observe the change of the prediction similarity and take the value of ω when the similarity is the highest.

Algorithm 1 Automatic Detection of weight ω

Input: Bug reports training set B1 = {$B1_i$}; Bug reports testing set B2 = {$B2_i$}; Step size ε ; Developer Set D = {D_i};
Output: The weight ω

```
1.  Calculate SimilarityW and SimilarityT in B1 and B2
2.  maxcount = 0;
3.  for each Δ∈[0,1] with a step size ε, do
4.     for each B2_i ∈B2
5.        count = 0;
6.        get B1_j with the biggest Similarity(B1_j,B2_i)
7.        if (D_i = D_j) then
8.           count ++;
9.        end if
10.    end for
11.    if (count > maxcount) then
12.       maxcount = count; ω = Δ;
13.    end if
14. end for
15. return ω
```

2) Online Recommendation Stage

After we get the weight ω in the offline learning stage, we can carry on to the online recommendation stage. In this stage, ω is used to get the similarity vector between bug reports. With ω adjusting weights, the similarity between bug report a and u *Similarity(a,u)* is

$$Similarity(a,u) = \omega SimilarityW(a,u) + (1-\omega)SimilarityT(a,u). \quad (4)$$

Similarity (a,u) ranges between 0 and 1. It implies ascending similarity with its value changing from 0 to 1. With *Similarity(a,u)*, we can get the accuracy vector:

$$Accuracy(u) = (accuracy_1, \ldots, accuracy_i, \ldots, accuracy_n) \quad (5)$$

$$accuracy_i = Similarity(a_i, u). \quad (6)$$

Each bug report has a time cost attribute which we use as the normalized cost of the bug report to form the cost vector, which is

$$Cost(u) = (cost_1, \ldots, cost_i, \ldots, cost_n). \quad (7)$$

where $cost_i$ is calculated as

$$cost_i = \frac{1/time_i}{1/\min(Cost(u))} \quad (8)$$

$time_i$ is the time bug report i used to be fixed.

With these two vectors, the portfolio theory is applied to strike a balance between accuracy and cost to make the most appropriate recommendation by assigning a weight to each of the bug report. Ordering the bug reports by the weight ascendingly gets us the final recommendation result.

The detailed algorithm is shown in Algorithm 2.

Algorithm 2 Automatic Developer Recommendation

Input: Bug reports training set B = {B_i}; New bug report b;
Output: The recommended developer D

1. Take bug reports from B with severity level higher than b as training set as B'
2. Take each bug report in B' and get its cost and Accuracy to the new bug report as Cost ={$Cost_i$} and Accuracy={$Accuracy_i$} , respectively
3. Apply modern portfolio theory and get a weight vector W={w_1, w_2, …, w_n} with w_i as the i^{th} bug report in B'
4. Get the bug report with the largest w and take its fixer as D
5. return D

C. Using Modern Portfolio Theory to Rank

When a new bug report comes up, it's useless to take bug reports with lower severity level as candidate recommender, for that the developer it recommends is highly impossible to fix the bug in time. So we take a strategy to filter out bug reports with lower severity level. This action is also beneficial for the instantaneity of recommendation. There are two types of ranking principles for recommendation.

- *Accuracy Principle*: We first rank bug reports in ascending order by their accuracy, and bug reports with the same accuracy will be further ranked by their cost.
- *Cost Principle*: We first rank bug reports in descending order by the cost, and bug report having the same cost will be further ranked by accuracy.

To make a finer recommendation structure, we need strike a balance between these two attributes, and thus modern portfolio theory [21] is used. This theory is first introduced in the field of portfolio investment. For example, there are n stocks; each stock has a future return with a risk. The theory aims at acquiring more future return with lower risk. In our problem, bug reports can be taken as the stocks, their accuracy can be regarded as the future return and their cost as risk. In specific, this theory assigns a weight to each of the bug reports which can be used to rank the bug reports. The bug report with the biggest weight should be the recommender.

Specially, a bug report portfolio can be represented by a collection of n bug reports with a corresponding weight assigned to each bug report b, i.e.

$$\gamma=\{(a_i, \omega_i)\}, \qquad \text{s.t.} \sum_i \omega_i = 1. \quad (9)$$

The weight ω in our problem indicates how much attention the recommendation system wants the user to pay on the bug report b_i. Therefore, the weights can be used to determine the ranks of bug reports; that is, bug reports should be ranked by the descending order of their weights. The future return of bug reports is E[γ], which can be computed by

$$\mathrm{E}[\gamma] = \sum_i^n \omega_i \cdot \Delta_i^{-1} \cdot \quad (10)$$

where Δ_i is the rank of bug reports b_i in the accuracy based rank list. Also, the future risk of bug reports is defined as R[γ], which can be computed as

$$\mathrm{R}[\gamma] = \sum_i^n \left(\omega_i^2 \nabla_i^{-2} + 2 \sum_{j=i+1}^{n} \omega_i \omega_j \nabla_i^{-1} \nabla_j^{-1} J_{ij} \right). \quad (11)$$

where ∇_i is the rank of bug report b_i in the cost based ranked list, and J_{ij} is the risk correlation between Apps b_i and b_j. Here, we estimate J_{ij} as *Similarity(b_i, b_j)*.

In our problem, the objective is to learn a set of bug report weights w for maximizing the accuracy and minimizing the cost, i.e.

$$\arg\max_w \left(E[\gamma] - b \cdot R[\gamma] \right). \quad (12)$$

where b is a specified risk preference parameter, which can be defined as the given severity level in our experiments. The simplest way is to set b to a default value of 1, which equalizes the importance of the accuracy and the cost. We leave method defining the value of b under different circumstances to future work. In the experiment, we vary the value of b to see the relationship between accuracy and cost.

IV. Experimental Setup

In this section, we evaluate the developer recommendation system with awareness of accuracy and cost (DRAC) approach with a large-scale real-world dataset.

A. Data Collection

This section presents our analysis on data we acquire and discuss the rationality of their appearance. We collect 484,870 bug reports from the project 'Eclipse' which is publicly accessible on the open source platform-*Bugzilla*. Those bug reports have different status. The status of bugs includes: unconfirmed, new, assigned, resolved, verified, closed and reopened. They represent different phases of bug fixing process. Only fixed bugs appear valid in the experiment. So we filter out non-fixed bug reports and that leaves 265,280 of them.

We mainly use the description of the bug reports by turning them into word count vectors and topic possibility vectors. First, we preprocess description by stemming, removing stop words, numerals and words with length over 20 characters. After preprocessing, bug reports become a set of words. To get the word count vector, dictionary is formed, and we take the count of the word in each bug report as the value of that word. For the topic vector, we apply LDA to the words to extract the corpus' topics and then calculate bug reports' possibility belonging to each of the topics as the topic vector. In this pro-

cess, we adopt a widely used LDA implementation LDA4j[5]. We followed guides in [22] and set the number of topics to 17. The format of the topic is shown in TABLE 1. Each topic has a set of words which correspond to its possibility of appearing in this topic. We take the square error of word frequency as the similarity between a bug report and a topic.

TABLE I. EXAMPLE WORD DISTRIBUTION OF TOPICS

Topic 1		Topic 14		Topic 17	
property	0.043	org	0.042	file	0.092
persist	0.027	java	0.020	project	0.066
connect	0.018	eclipse	0.014	build	0.032
test	0.015	report	0.013	plugin	0.027
map	0.014	birth	0.012	create	0.022
query	0.013	apache	0.010	jar	0.018
jpa	0.010	engine	0.009	package	0.018
value	0.010	service	0.009	path	0.016
service	0.009	jetties	0.009	folder	0.014
session	0.008	invoke	0.009	workspace	0.013

B. *Baseline: Costriage*

To our best knowledge, there is only one similar research [23]. It brought up the method *Costriage*. *Costriage* follows three steps. First, it constructs the developer profiles, which is a numeric vector with each element denotes the developer's estimated cost L_C for fixing a certain bug type. To fill in the blanks in developer's profile, collaborative filtering is used to predict the missing value. Thus, it gets the developers' cost to solve every bug type. Second, it trains a multi-class classifier, when a new bug came, it estimates each developer's score for the new bug and form the vector L_S by extracting its feature vector and applying the classifier. L_C and L_S are

$$L_C = \left(s_{C[1]}, s_{C[2]}, \cdots, s_{C[n]}\right), L_S = \left(s_{S[1]}, s_{S[2]}, \cdots, s_{S[n]}\right). \quad (13)$$

where $s_{c[i]}$ means the i^{th} developer's estimated cost to fix the given bug, and $s_{s[i]}$ denotes the i^{th} developer's success possibility for fixing the new bug.

Third, L_C and L_S are merged into one vector L_H, and then the first developer is assigned to the new bug report.

$$L_H = \left(s_{H[1]}, s_{H[2]}, \cdots, s_{H[n]}\right) \quad (14)$$

$S_{H[i]}$ is obtained by

$$s_{H[i]} = \alpha \cdot \frac{s_{S[i]}}{\max\left(L_S\right)} + \left(1-\alpha\right) \cdot \frac{1/s_{C[i]}}{1/\min\left(L_C\right)}, \quad (15)$$

where $0 \le \alpha \le 1$.

V. EXPERIMENTAL RESULTS

A. *Bug Report Severity Levels*

To see the relationship between topics and severity levels, we do statistics about the distribution of severity level with different topics. Fig. 3(a) shows the percentage of bug reports with different topics. The bug reports belonging to topic 14 take up the majority of bug report repository. Topic 10, 11, 17 is the following topic with relatively large volume. Fig. 3 (b)-(d) shows the percentage of bug reports with different severities in topic 3, 11 and 14, respectively. Based on our results, "Normal" bug reports is the main component of all the topics with a proportion fluctuating around 70%. It would be hard to see other severity levels' status with "Normal" in the figure. So we leave out bug reports with severity level "Normal" and clarify other severity levels' distribution among topics. In these figures, we can see that topics have their bias towards different severity level. Topic 14, which has the most bug reports, contains bug reports with severity level 'Enhancement' and 'Major' mainly. Half bug reports belonging to Topic 11 are 'Major' which is much higher than the other two topics. And Topic 3 is mainly composed of 'Enhancement' bug reports. Different topics have slight bias towards different severity levels. Based on the statistical results, we can see that there is a correlation between topic and severity level preference. Different topics have different severity level components.

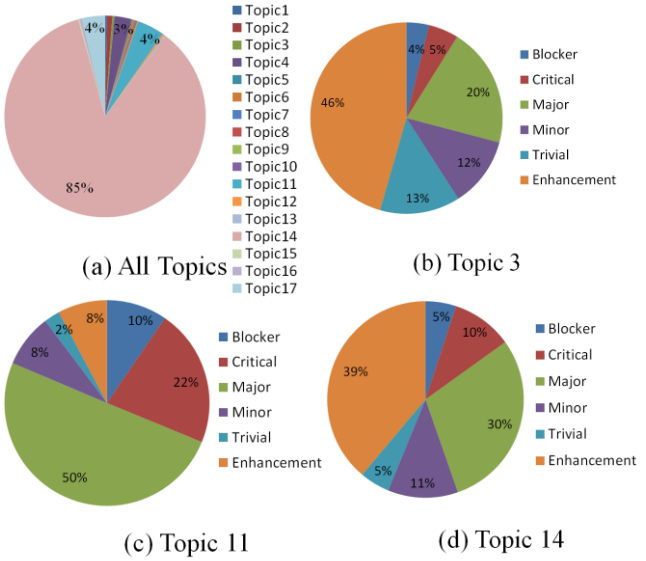

Figure 3. The percent of bug reports (a) and bug report topics at different severity levels (b)-(d)

B. *Evaluation of Developer Recommendation*

In the offline learning stage, to learn the weight between word vector and topic vector, we set the step length to 0.2. And accuracy spikes with 0.6. So we set the value of ω to 0.6. We order the bug reports in time series, and take the first 80% of them as training set with the remaining 20% as testing set. Specifically, we set up the evaluation as follows. First, we implement *DRAC* and *Costriage*. Then, we observe the trade-offs between bug assignment accuracy and bug fix time using DRAC compared to *Costriage*. We apply them with varying b to observe the trade-offs between accuracy and average bug fix time. Fig. 4 shows the comparison result. The x-axis represents the average time to fix one bug, and the y-axis represents bug assignment accuracy.

We can see from Fig. 4 that there is an obvious sign of trade-off. When the accuracy of the recommendation ascends, the average time to fix a bug increases with it, which is very understandable and confirms the idea we brought before. By altering the weight b, the developer can get different recommendation results with different purposes.

[5] https://github.com/hankcs/LDA4j/tree/master/src

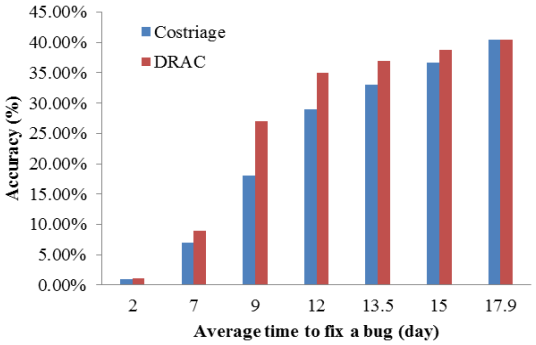

Figure 4. The trade-offs between accuracy and bug fix time

VI. CONCLUSION

In this paper, we developed a reviewer recommender system with awareness of accuracy and cost. Specifically, we learned a function to determine the weight of data from different sources. Moreover, to consider both reviewers' accuracy and cost, we introduced a flexible recommendation method based on modern portfolio theory. A key challenge in this research field is upholding the recommendation speed with big volume and high growth rate of the data. To address this problem, we propose *DRAC* to reduce the size of dataset and make recommendation considering the severity level and importance of bug reports. The experiments on a large-scale real-world dataset clearly clarified the effectiveness of the proposed recommendation framework.

Our recommendation framework is not restricted to developer recommendation. It can be applied to other field with similar objective. Take code fragment recommendation as example, it needs to consider both the applicability of the recommended code fragment (accuracy) and its security. These two factors correspond to the two objectives we have in this problem. So for problems with two optimization objectives like code recommendation, our recommendation algorithm is theoretically applicable to them.

REFERENCES

[1] J. Anvik, L. Hiew, and G. C. Murphy, 2006. "Who should fix this bug?" In ICSE'06.

[2] J. Zhang, X. Y. Wang, D. Hao, "A survey on bug-report analysis"[J]. Science China, 2015, 58(2):1-24.

[3] Z. Xu, Y Liu, Y. Xuan J, et al. Crowdsourcing based social media data analysis of urban emergency events[J]. Multimedia Tools & Applications, 2015:1-18.

[4] Z. Xu, Y. Liu, N. Y. Yen, et al. Crowdsourcing based Description of Urban Emergency Events using Social Media Big Data[J]. IEEE Transactions on Cloud Computing, 2016:1-1.

[5] T. T. Nguyen, A. T. Nguyen, T. N. Nguyen, "Topic-based, time-aware bug assignment"[J]. Acm Sigsoft Software Engineering Notes, 2014, 39(1):1-4.

[6] T. Zhang, H. Jiang, Luo X, "A Literature Review of Research in Bug Resolution: Tasks, Challenges and Future Directions"[J]. Computer Journal, 2015.

[7] S. Banerjee, J. Helmick, Z. Syed, "Eclipse vs. Mozilla: A Comparison of Two Large-Scale Open Source Problem Report Repositories"[C]// High Assurance Systems Engineering (HASE), 2015 IEEE 16th International Symposium on. IEEE, 2015:263-270.

[8] C. L. P. Chen, C. Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on Big Data[J]. Information Sciences, 2014, 275(11):314-347.

[9] E. Savitz, Gartner, "10 Critical Tech Trends for the Next Five Years", October 2012.<http://www.forbes.com/sites/ericsavitz/2012/10/22/gartner-10-critical-tech-trends-for-the-next-five-years/>.

[10] J. Xuan, H. Jiang, Y. Hu, "Towards Effective Bug Triage with Software Data Reduction Techniques"[J]. IEEE Transactions on Knowledge & Data Engineering, 2015, 27(1):264-280.

[11] P. Nagarnaik, A. Thomas, "Survey on recommendation system methods." In Electronics and Communication Systems (ICECS), 2015 2nd International Conference on, pp. 1496-1501. IEEE, 2015

[12] Z. Bo, N. Lulian, G. A. Rajiv, "Cross-platform Analysis of Bugs and Bug-fixing in Open Source Projects Desktop vs. Android vs. iOS." In: Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering, Nanjing, China, 2015.

[13] J. Xuan, H. Jiang, Z. Ren, "Automatic bug triage using semi-supervised text classification." In: Proceedings of International Conference on Software Engineering & Knowledge Engineering, Redwood City, 2010. 209–214

[14] G. Canfora, L. Cerulo. "Supporting change request assignment in open-source development." In: Proceedings of the ACM Symposium on Applied Computing, Dijon, 2006. 1767–1772

[15] D. Matter, A. Kuhn, O. Nierstrasz, "Assigning bug reports using a vocabulary-based expertise model of developers." In: Proceedings of the International Working Conference on Mining Software Repositories, Vancouver, 2009. 131–140

[16] G. Jeong, S. Kim, T. Zimmermann, "Improving bug triage with bug tossing graphs." In: Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, Amsterdam, 2009. 111–120

[17] A. Tamrawi, T. T. Nguyen, J. Al-Kofahi, "Fuzzy set-based automatic bug triaging." In: Proceedings of the International Conference on Software Engineering, Waikiki, 2011. 884–887

[18] X. Xia, D. Lo, X. Wang, "Accurate developer recommendation for bug resolution." In: Proceedings of the Working Conference on Reverse Engineering, Koblenz, 2013. 72–81

[19] H. Hu, H. Zhang, J. Xuan, "Effective bug triage based on historical bug-fix information." In: Proceedings of the IEEE International Symposium on Software Reliability Engineering, Naples, 2014. 122–132

[20] M. Schulmerich, Y. M. Leporcher, C. H. Eu, "Modern Portfolio Theory and Its Problems"[M]// Applied Asset and Risk Management. Springer Berlin Heidelberg, 2015:101-173.

[21] J. Wang and J. Zhu, "Portfolio theory of information retrieval." In *Proceedings of the 32Nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '09, pages 115{122, New York, NY, USA, 2009. ACM.

[22] R. Arun, V. Suresh, C. E. V. Madhavan, M. N. N. Murthy, 2010. "On finding the natural number of topics with latent dirichlet allocation: Some observations." In PAKDD'10

[23] J. W. Park, M. W. Lee, J. Kim, "CosTriage: A Cost-Aware Triage Algorithm for Bug Reporting Systems."[C]// Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011. 2011.