A Bug Assignment Technique Based on Bug Fixing Expertise and Source Commit Recency of Developers

Afrina Khatun*, Kazi Sakib†
Institute of Information Technology, University of Dhaka, Bangladesh
*bit0411@iit.du.ac.bd, †sakib@iit.du.ac.bd

***Abstract*—Automatic bug assignment is an essential activity aiming at assigning bugs to appropriate developers. Existing approaches consider either recent commits or previous bug fixes of developers, leading to recommendation of inexperienced or inactive developers respectively. Considering only one information source leads these approaches to low prediction accuracy. An approach called ERBA is proposed, which considers both expertise and recent activities of developers. ERBA first processes source code and commit logs to construct an index connecting the source entities with developer recent activities. Next, it takes fixed bug reports and builds another index, mapping the bug report keywords with developer bug fixing expertise. On arrival of new bug reports, the final module queries the two indexes using the new bug report terms, and applies tf-idf technique on the query result to calculate an ERBA score for developers. Finally, an ascending ordered list on ERBA score is suggested. For assessment of competency, a case study has been conducted on Eclipse JDT. It depicts that ERBA outperforms existing approach by improving prediction accuracy from 33.8% upto 44%. The result also represents that ERBA shows the first correct developer on average near 4.04 ranks, whereas existing approach shows in 7.27.**

***Keywords*—Bug assignment, Bug reports, Term weighting technique, Recommendation**

I. Introduction

As software bugs are inevitable, bug assignment is an essential step for fixing. Bugs in open source projects are maintained and accumulated using bug tracking systems (e.g. Bugzilla). In case of large scale projects, bug tracking systems receive 50 to 60 bugs on a daily basis [1]. One important part of bug assignment is finding an appropriate developer for newly arrived bugs. The developers in open source software systems, generally are huge in number, and work parallel in various project modules. So finding an appropriate developer is a difficult task. Manual assignment of bug reports is error-prone and time consuming. For example, mistake in assignment process usually lead to unnecessary bug reassignments, and slow down the bug fixing process. In this case, automatic and accurate bug assignment can facilitate bug triager with potential list of bug fixers for newly arrived bugs.

The main task of bug assignment is finding appropriate developers by analyzing available information sources - narrative bug reports, source code and commit logs. Bug assignment can be done through analysing previously fixed bug reports. With passage of time, team or company switch of developers may occur, which leads to recommendation of inactive developers. Analysing software source code related commits for identifying recent developers is another solution of bug assignment. Although this approach can prevent recommendation of inactive developers, it totally ignores the developers experienced in fixing similar bug reports. Due to considering only recent source commits, this approach can recommend novice or inexperienced developers. In order to assess an appropriate developer for a newly arrived bug, both experienced and recent developer needs to be considered. Considering only one of the information can not compensate the other resulting in inaccurate developer recommendation.

Understanding the importance of automatic bug assignment, various techniques have been proposed in the literature. Automatic bug assignment was first formulated as machine learning problem by using text categorization [2]. BugFixer, a developer recommendation method has been proposed by Hao et. al [1]. This method constructs a Developer-Component-Bug (DCB) network, using past bug reports, and recommends developers over this DCB network. This recommendation list becomes less accurate with joining or switching of developers. TRAM, another bug triaging approach which considers attributes such as reporter, component and discriminating terms of bug report as an addition to the traditional approaches for training the recommendation model [3]. This technique also suffers from the above mentioned limitations of considering historical activities. A code authorship based recommendation technique has been proposed by Mario et. al which first indexes source code entities to identify buggy files [4]. It then recommends the authors of buggy portion as bug fixer. Shokripour et. al [5] have proposed a time based approach that indexes source identifiers along with commit time to prioritize the recent developers for recommendation. However, both of these techniques fail to achieve high accuracy due to ignoring developer experiences.

An automatic bug assignment technique, considering previous bug fixing expertise and recent source commits of developers called ERBA (Expertise and Recency based Bug Assignment), has been proposed in this paper. The *Recency Determination*, *Expertise Determination* and *Developer Recommendation* modules operate together to support the technique implementation. The Recency Determination module takes source code and commit logs as inputs. To represent the time of identifier usage, it builds an index connecting source code identifiers with developer commits. Besides, the Expertise Determination module uses fixed bug reports to construct an index connecting bug report features (keywords) with the report fixer. Finally, when new bug reports arrive, the Developer Recommendation module extracts the bug report keywords, and queries these keywords on the indexes built by the above mentioned modules. By applying tf-idf term weighting technique on the query results, a score is assigned, called ERBA score to each developer. The high scored developers are recommended as appropriate fixers.

A case study on an open source project, Eclipse JDT has been conducted for assessment of ERBA. The commit history

and bug reports of JDT has been collected from open source Git and Bugzilla repositories respectively. Total 25000 commit logs [5] and 7600 bug reports have been taken under consideration. For evaluating competency, ERBA has been compared with an existing bug assignment technique, ABA-time-tf-idf [5]. A test set of 100 bug reports are taken, which is also used in [5] and applied on both ERBA and ABA-time-tf-idf for evaluation. The result depicts that ERBA has successfully improved the ranking accuracy from 33.8% to 44%. Besides, a decrease in the average position of first correct developer from 7.27 to 4.04 indicates the increased effectiveness of ERBA.

II. Related Work

Concerned with the increased importance of automatic bug assignment, a number of techniques have been proposed by researchers. For recommendation, some of the existing techniques use previous fixes by developers. Again, some of the techniques consider current activities of developers for identifying actual fixers. A couple of significant works related to this research topic are outlined in the following.

A survey on various bug triaging techniques has been presented by Sawant et. al [6]. The survey divided existing bug triaging techniques into text categorization, tossing graph, cost aware algorithm, software data reduction and source based techniques etc. Text categorization based techniques build a model that trains from past bug reports to predict correct rank of developers ([1],[2],[3],[4],[7]). Baysal et al. has enhanced the text categorization techniques by adding user preference of fixing certain bugs in recommendation process [2]. The framework performs its task using three components. The *Expertise Recommendation* component creates a ranked list of developers using previous expertise profile. The *Preference Elicitation* component collects and stores the preference level of fixing certain bug types through a feedback process. Lastly, knowing the preference and expertise of each developer, *Task Allocation* component assigns bug reports. Since the framework considers only past historical activities, it ignores the current activities of developers, and recommends developers who are either working in another project or company. As a result, inactive developers are recommended which reduces prediction accuracy.

Tossing graph based bug triaging techniques for reducing bug reassignment has also been developed by researchers (e.g. [8], [9], [10]). The main focus of these techniques is to reduce the number of passes or tosses a bug report goes through because of incorrect assignment. In such techniques, the graph is constructed using previous bug reports ([9], [10]). Therefore, as mentioned above, these historical activities lead to low accuracy of the recommended list. A fine grained incremental learning and multi feature tossing graph based technique has been proposed by Bhattacharya et. al [8]. It improves the previous techniques by considering multiple bug report features like product and component while graph construction. Due to considering previous information, the technique results in search failure in case of new developer arrival.

CosTriage, a cost aware developer ranking algorithm has been developed by Park et. al [11]. The technique converts bug triaging into an optimization problem of accuracy and cost, which adopts Content Boosted Collaborative Filtering (CBCF) for ranking developers. As the input to the system is only previous bug history, the technique recommends inactive fixers.

Few data reduction techniques for effective bug triaging has also been proposed in the literature (e.g. [12], [13], [14]). Xuan et al. combine existing techniques of instance (such as duplicate bug report) selection and feature (such as keywords) selection to simultaneously reduce the bug and the word dimensions [14]. The technique uses a feature selection and instance selection algorithms on existing bug repository. The reduced bug data contains fewer bug reports and fewer words than the original bug data and provide similar information over the original bug data. The major flaw of data reduction based techniques is that the reduced data set is another representative of historical developer activity based information.

Few source based bug assignment techniques have also been proposed by researchers. Matter et. al has suggested Develect, a vocabulary based expertise model for recommending developers to assign bugs [7]. The model parses the source code and version history for indexing bag of words representing vocabulary of each source code contributor. A model is trained using existing vocabularies and stored in a matrix as a term-author-matrix. For new bug reports, the model checks the report keywords using lexical similarities against developer vocabularies. The highest scored developers are taken as fixers. For overcoming inactive developer recommendation, the technique uses a threshold value. However, the technique totally ignores experienced developers, which leads bug assignment to novice or inexperienced developers. Subsequently, it increases bug reassignments, prolongs fixing time and reduces recommendation accuracy.

Another source based bug assignment technique has been proposed in [5]. While identifying the recent developers, this technique focuses on using time meta data. The technique first parses all the source code entities (such as name of class, method, method parameters and class attributes) and connects these entities with contributor to construct a corpus. In case of new bug reports, the keywords are searched in the index and given a weight based on frequent usage and time metadata. As a result, the more recent a term is used by a developer, the more emphasize the developer gets. The high scored developers are shown at the top of the list. Although it correctly identifies recent developers, it generally fails to achieve high accuracy due to ignoring experienced developers.

Various techniques for automatic fixer recommendation have been proposed in the literature. Most of the techniques learn from the previous fix history (e.g. [1],[2],[3],[4]) or current commit history (e.g. [5],[7]). However, these techniques suffer from low accuracy due to recommendation of inactive or inexperienced developers respectively. Considering only one of the information source cannot improve the prediction accuracy.

III. Methodology

To achieve improved fixer recommendation accuracy, an automatic bug assignment technique called ERBA, is proposed in this section. As stated before, bug solving history is an important factor in developer expertise determination while bug triaging process. On the other hand, due to factors such as team switching and working in different project modules, the recency of developer activities also plays significant role in bug assignment. Therefore, an automatic approach is required that combines these information sources. The detailed developer recommendation procedure of ERBA is illustrated below.

A. Overview of ERBA

ERBA performs developer recommendation in three steps. The overview of ERBA technique is shown in Fig. 1. The *Recency Determination* module is responsible for extracting developer current activities. Developer commits regarding source code changes generally reveals their recent activity information. So this module uses software source code and commit logs for determining developer recency. The *Expertise Determination* module is liable for collecting expertise in solving similar bugs. This module reads software bug tracking history and identifies developers who have previously fixed relative bug reports. Finally, the *Developer Recommendation* module performs ultimate developer rank suggestion for a new bug report. It combines developer recency and expertise gained from those two modules, and assigns an ERBA score to each developer based on which developers are ranked.

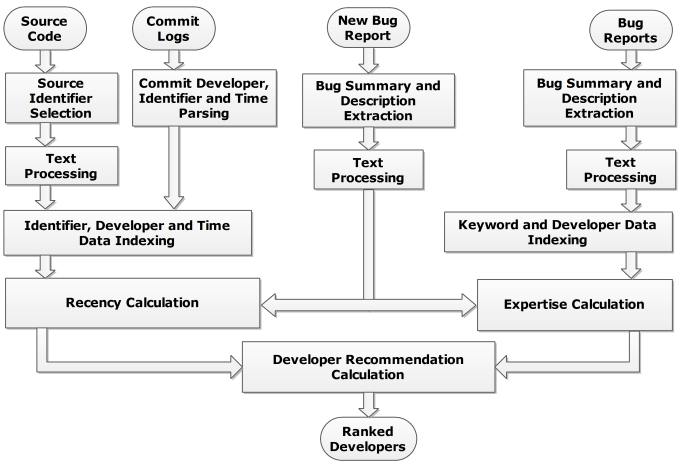

Fig. 1: Overview of the ERBA

B. Recency Determination

Determining developer's recent activities while bug assigning, helps to eliminate inactive developer considerations. Therefore, the *Recency Determination* module of ERBA is responsible for identifying developer recent activities. This module takes software source code and commit log as inputs as shown in Fig. 1. It then parses and extracts the names of class, method, attribute and method parameters from each source file. These data are taken into consideration as the source code entities representing developer vocabularies and activities. Moreover, data by nature may contain unnecessary characters and keywords. For improved tokenization, all these parsed data undergoes through text processing. The text processing step contains identifier decomposition based on CamelCase letter and separator character, number and special character removal, stop word removal and stemming. For retrieving relevant results, both the complete and decomposed identifiers are considered while indexing. Besides, the commit history of the project is extracted in XML format using git commands. The extracted XML contains commit id, author, date, subject and a list of identifiers associated to the commits.

In order to determine developer's recency of using identifiers, proper links between the extracted source identifiers and commit logs need to be established. Again an identifier can be used by a number of developers in different phase of time during the project development. Therefore, when a new bug report arrives, an index of all the identifiers need to be built for searching recent developers. A posting list needs to be constructed that maps the identifiers with the developers who used those. So, following Algorithm 1 is proposed which performs the task of developer current activity collection. Algorithm 1 takes a

Algorithm 1 Recent Activity Collection

Input: A list of string identifiers (I)
Output: An index associating identifiers with a postinglist of developers, ($DeveloperActivity$). Each $DeveloperActivity$ represents a data structure containing developer name ($name$) and identifier usage time ($time$)

```
1:  Begin
2:  Map < String, List < DeveloperActivity >> postingList
3:  DeveloperActivity d
4:  for each i ∈ I do
5:      Commits ← getCommitsOfIdentifier(i)
6:      for each c ∈ Commits do
7:          d ← new DeveloperActivity()
8:          d.name ← c.author
9:          d.time ← c.date
10:         if !postingList.keys.conatins(i) then
11:             postingList.keys.add((i)
12:         end if
13:         postingList[i].developers.add(d)
14:     end for
15: end for
16: End
```

processed list of identifiers as input. In order to associate the list of identifiers with corresponding developer list, a complex data structure, *DeveloperActivity* is constructed. It contains two properties - developer name and commit time. To construct the index, initially an empty *postingList* is declared (Algorithm 1, line 2). An empty variable, *d* of type *DeveloperActivity* is also declared for populating the *postingList* later in line 3. A *for* loop is defined to iterate through the identifier list, *I* for index construction (Algorithm 1, line 4). For each identifier, *i* corresponding commits in which the identifier is found, are extracted by calling function *getCommitsOfIdentifier* as shown in line 5. This function takes an identifier as input and returns a commit list of type *Commit*. Each *Commit* contains aforementioned commit log attributes. A nested inner loop to iterate on the *Commits* list is also defined (Algorithm 1, line 6). Each iteration of this loop initializes *d* with a new *DeveloperActivity* instance. It then updates the *name* and *time* property of *d* with *author* and *date* property of commit, *c* respectively (Algorithm 1, line 7-9). For adding identifiers into the *postingList*, the list is first checked whether it already contains the identifier (Algorithm 1 line 10-12). In the next step, the updated value of *d* is added to the *postingList* against the identifier, so that when searched, the list of associated developers can be fetched (Algorithm 1, line 13). Finally, Algorithm 1 returns an index of triplet data structure containing identifiers, developers who use those identifiers and their identifier using time.

C. Expertise Determination

The number of related fixed bugs by a developer is another developer recommendation indicator. The *Expertise Determi-*

nation module of Fig. 1 performs the task of identifying expert developers, who are proficient for fixing similar bugs. For this purpose, the bug reports in XML format are collected from Bugzilla and inputted in the *Expertise Determination* module. These bug reports are also collected in XML format for making it program readable. The bug repository contains either of two types of bugs - open and closed. For identifying expert developers, the technique requires to index bug reports which are already fixed. Therefore, ERBA takes closed bug reports into consideration. The closed bug reports which have RESOLVED and VERIFIED as bug status, and FIXED as bug resolution are selected. Again, the summary and description properties of a bug report represents bug features, and so are taken by ERBA for further indexing.

The more familiar a developer is with the terms of fixed bug reports, the more expertise the developer has in fixing these term related bugs. In order to determine developer's expertise, an index of terms needs to be built for searching developers who have fixed these term related bugs. As a term related bugs can be fixed by several developers, a postinglist is required which connects the terms with these list of developers. For this purpose, a function similar to Algorithm 1 is used. The function is inputted with a list of fixed bug reports. Each bug report has two properties - a list of bug report keywords and the name of a developer who fixed the bug. The list is constructed using the keywords of bug summary and description. All the members of the list go through the same text processing step discussed earlier. The mappings between bug report terms and bug fixer are maintained in a postinglist. At the end, the function constructs another index as output containing developer expertise information.

TABLE I:
Explanation of Attributes & Methods of Algorithm 2

Variables	
useFreq	no. of times a term is used in source commits
fixFreq	no. of times a term related bug is fixed
Date	last usage time of a term
#dev	no. of developers in the project
Methods	
devUseFreq(t)	takes term, t as input and returns the no. of developers commit the term
devFixFreq(t)	takes term, t as input and returns the no. of developers fixed the term related bugs

D. Developer Recommendation

Finally, ERBA combines both the recency and expertise information gained from above mentioned indexes for appropriate developer ranking as shown in Fig. 1. When a new bug report arrives, a search query is formulated using the summary and description field of the report. This query is executed on the two indexes built by the *Recency* and *Expertise Recommendation* modules. Both the queries return a set of developers who uses or fixes the term of the new bug report. This module then combines these query results and construct a complex data structure of type *Map<String,Map<String,TermInfo>>>*. The outer map associates each developer with a list of new bug report terms used by this developer. On the other hand, the inner map connects each term, to its usage information of type *TermInfo*. *TermInfo* represents a data structure consists of three properties - *useFreq*, *fixFreq* and *Date*. The explanation of these properties is shown in Table I.

ERBA performs ultimate developer recommendation using

Algorithm 2 Developer Recommendation

Input: A new bug report (B). (B) contains a set of keywords ($terms$) and bug reporting time ($Date$).
Output: A sorted developer list ($devList$).

1: **Begin**
2: $Map < String, Map < String, TermInfo >>$ $devTermInfo \leftarrow$ $getDeveloperTermUsageInformation(B)$
3: $List < Developer > devList$
4: **for** $d \in devTermInfo$ **do**
5: $double\ recency, expertise$
6: **for** $term \in devTermInfo[d.name]$ **do**
7: $t \leftarrow devTermInfo[d.name][term]$
8: $TfIdf \leftarrow t.useFreq * log(\#dev/devUseFreq(t))$
9: $time \leftarrow (1/devUseFreq(t))+$ $(1/\sqrt{(B.Date - t.Date)})$
10: $recency+ \leftarrow tfIdf * time$
11: $TfIdf \leftarrow t.fixFreq * log(\#dev/devFixFreq(t))$
12: $expertise+ \leftarrow tfIdf$
13: **end for**
14: $dev = newDeveloper()$
15: $dev.name \leftarrow d.name$
16: $dev.score \leftarrow recency + expertise$
17: $devList.add(dev)$
18: **end for**
19: $devList.sort()$
20: **End**

Algorithm 2. It takes a new bug report (*B*) as input. The *getDeveloperTermUsageInformation* function is inputted with this bug report. It then performs the above mentioned search query formulation and execution task (Algorithm 2, line 2). As a result, it returns complex data structure *devTermInfo* constructed from query results. An empty list of developers, *devList* of type *Developer* is declared in line 3 for storing ultimate developer rank. An outer loop is defined at line 4 to iterate on *devTermInfo*. Two empty variables *recency* and *expertise* are declared for each developer information *d* in *devTermInfo* (line 5). An inner loop is constructed to iterate over the new bug report terms used by this developer (line 6). For each *term*, its corresponding term usage information *t* of type *TermInfo* is extracted from *devTermInfo* (line 7).

For accurate developer recommendation, the recent activities is an important factor. tf-idf term weighting technique is applied for measuring the frequent usage of a term by a developer (line 8). This technique determines the weight of a term using the frequency of its usage (*t.useFreq*) and the generality of it in the project (*log(#dev/devUseFreq(t))*). Considering only frequent use of developer may result in recommendation of inactive developers. To alleviate this problem, the recent use of a term is incorporated with its frequent use. The recency of a term is determined by adding the inverse of the number of developers used this term, and the inverse of time difference between the bug reporting time ($B.Date$) and this term using time ($t.Date$) (line 9). The greater the value of *devUseFreq(t)* is, the more important the term is for the developer. Again the

greater the time interval (*B.Date-t.Date*) is, the less recent the term is used. This recency of a term (*time*) is then multiplied by its frequency (*tf-Idf*) to calculate developer's recency on this term. Finally, the sum of recency of all terms determines the developer recency (line 10).

On the other hand, the expertise of a developer in each term is calculated in similar tf-idf technique using term fixing frequency (*fixFreq*) and the generality of fixing the term related bugs (*log(#dev/devFixFreq(t))*) (line 11). Developer's expertise is determined by summing the expertise of fixing all terms (line 12). Lastly for each iteration of *devTermInfo*, an instance, *dev* of type *Developer* is constructed, initialized and inserted into *devList* (line 14-17). This instance is initialized using developer name and an ERBA score calculated by summing the recency and expertise score of each developer. The technique incorporates recency and expertise value for assigning ERBA score to developers. As a result, Algorithm 2 ends its task by sorting the *devList* in a descending order based on ERBA scores. ERBA concludes its task by suggesting the developers at the top of the list as appropriate developers for fixing the new bug.

IV. Case Study

For initial assessment of effectiveness, ERBA was applied on an open source project, Eclipse JDT [15]. JDT project is a set of plug-ins that adds the capabilities of a full-featured Java IDE to the Eclipse platform. This project is chosen due to following reasons. Firstly, this project has been used in various related approaches [5] for evaluation. Secondly, all information regarding source code, commit logs and bug reports of JDT are available in open source. Some of the properties about the JDT dataset which are taken into consideration are listed in Table II.

TABLE II: Properties of Eclipse JDT Dataset

FirstCommit	2001-05-03
LastCommit	2011-12-15
#Commits	51,425
#Java Files	6,899
#Bug Reports	7,600
#Developers	898

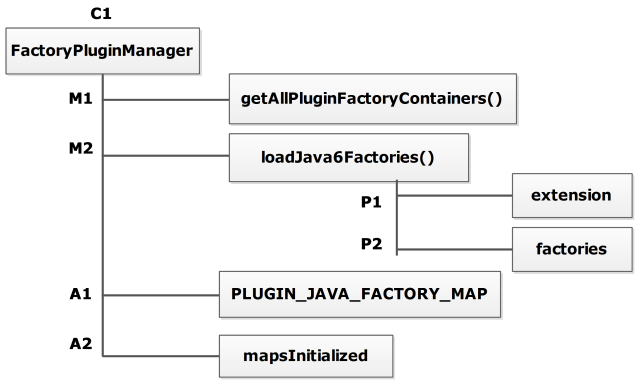

Fig. 2: Partial Extracted Structure of Class *FactoryPluginManager*

As stated before, the Recency Determination module takes Java source files and commit logs as input. It parses the source files and converts these into a data structure for identifier indexing. A data structure generated for class *FactoryPluginManager* of JDT source project is illustrated in Fig. 2. This module then processes the identifiers (such as names of class, method, method parameter and attribute) for improved tokenization. For example, it converts method name *loadJava6Factories* into *load*, *Java* and *Factori*. In order to connect these identifiers with appropriate developers, the commit log is extracted in XML format as shown in Fig. 3. The corresponding commits of each identifier are extracted by checking <*changedFiles*> tag. Moreover, associated developer name and identifier usage time are extracted from the <*author*> and <*date*> tags respectively. For example, as shown in Fig. 3 developer, *Dani Megret* and time, *Thu 15 Dec 2011 09:51:29* are inserted into the posting list of identifier, *Factori*.

```
<commit>
    <repo>eclipse.jdt.ui</repo>
    <hash>753aa8631e811e27271e75a7220f378b6a4e9fe5</hash>
    <author>Dani Megert</author>
    <authorMail>dmegert</authorMail>
    <date>Thu, 15 Dec 2011 09:51:29 +0100</date>
    <subject>Improved Javadoc</subject>
    <changedFiles>
        <identifier>IRefactoringHistoryControl</identifier>
        <identifier>FactoryPluginManager</identifier>
    </changedFiles>
</commit>
<commit>
<repo>eclipse.jdt.ui</repo>
```

Fig. 3: Partial Commit Log of JDT

On the other side, the *Expertise Determination* module extracts bug summary and description from the <*short_desc*> and <*thetext*> tags of XML formatted fixed bug reports respectively. The keywords of these bug reports are indexed against the bug fixer extracted from the <*assigned_to*> tag of XML. Finally when a new bug report arrives, the *Developer Recommendation* module starts processing. It queries the above built two indexes with the new bug report keywords. Based on the query results, this module associates an ERBA score to each developer as described in section III for ultimate developer suggestion.

For evaluating the competency of ERBA, it is compared with an existing time based bug assignment approach, ABA-time-tf-idf [5]. A dataset of 100 random selected bug reports has been used for checking ERBA recommendation validity. The validation of recommendation is analysed using two metrics - accuracy and effectiveness [5].

TABLE III: Comparison of Accuracy in percentage between ERBA and ABA-time-tf-idf

Approaches	**Top-1**	**Top-2**	**Top-3**	**Top-4**	**Top-5**	**Avg.**
ABA-Time-tf-idf	12	24	36	42	55	33.8
ERBA	15	28	47	62	68	44

Accuracy or Top N rank refers whether the top N recommended developers contain the actual bug fixer. If the top N developers contain the actual fixer, the recommendation is correct. A higher value of this metric represents higher accuracy of ERBA. For comparing the accuracy of ERBA with ABA-time-tf-idf, both these approaches were applied on the selected dataset. For each of these approaches, Top-1 to Top-5 recommendation accuray is listed in Table III. The

results clearly depicts that the proposed ERBA technique has higher accuracy than the existing technique for each of the Top-1 to Top-5 ranks. For example, ERBA has achieved 47% Top-3 recommendation accuracy whereas, ABA-time-tf-idf has correctly recommend the actual fixer within position three in 36% cases. From Table III it is found that, consideration of developer expertise along with their recency has enabled ERBA to improve the accuracy from 33.8% to 44%.

TABLE IV: Comparison of Effectiveness between ERBA and ABA-time-tf-idf

Approaches	Average Effectiveness
ABA-Time-tf-idf	7.27
ERBA	4.04

In case of automated bug assignment, effectiveness refers as the position of the first relevant developer in the ranking. Approaches that recommend relevant developers at the top of the list are considered more effective. It is so because ranking relevant developers at the top reduces the false positive results a triager needs to consider for recommendation. In this context, the lower the value of the metric, the less effort is conducted by the triager, leading to more effective recommender. The competency of ERBA with ABA-time-tf-idf in terms of effectiveness is calculated and listed in Table. IV. The results state that ERBA is suceessful in recommending the appropriate developer on average 4.04 position, which is lower than ABA-time-tf-idf (7.27). As stated before, a lower value indicates higher effectiveness. Therefore, accumulating both recent and previous activities of developers has created significant decrease in score, depicting increased effectiveness of ERBA. In short, for both of the metrics, ERBA performs better than ABA-time-tf-idf due to consideration of both experienced and recent developers.

V. Conclusion

Accurate assignment of bug reports to appropriate developers is an essential phase in software development. The existing approaches consider either previous bug fixes or recent source commits of developers in recommendation process. These individual approaches suffer from low prediction accuracy. While the previous activity based techniques recommend inactive developers, the current activity based techniques assign bugs to inexperienced developers. To overcome the limitations of individual approaches, a technique named ERBA is proposed which combines both historical fixing experience and current working activities to accurately recommend fixer list.

Three modules operate together to suggest appropriate developer ranking. While the *Recency Determination* module processes source code and commit logs to build an index of developer recent activities, *Expertise Determination* module identifies the mappings between developers and their fixing history, and stores these mappings in another index format. Finally, for new bug reports, *Developer Recommedation* module queries with bug report terms, and assigns an ERBA score to developers based on query result. Finally the developer list is ordered in ascending order and recommended as ulitimate bug fixers.

A case study on an open source project, Eclipse JDT is shown to analyze the competency of the technique. 25000 commit logs, 7600 bug reports and a test set of 100 bug reports were used for comparison with an existing technique named, ABA-time-tf-idf. The case study result shows that ERBA improves recommendation accuracy from 33.8% to 44%, and achieves increased effectiveness by identifying correct bug fixer (on average) at the position 4.04. In future, bug reports which are not previously fixed by any developers should be observed to identify how ERBA performs.

Acknowledgement

This work is supported by the fellowship from ICT Division, Bangladesh. No - 56.00.0000.028.33.065.16 (Part-1) -772 Date 21-06-2016. We would like to thank Fazle Mohammed Tawsif for his partial contribution in implementation of *Expertise Determination* module.

References

[1] H. Hu, H. Zhang, J. Xuan, and W. Sun, "Effective bug triage based on historical bug-fix information," in *25th International Symposium on Software Reliability Engineering*. IEEE, 2014, pp. 122–132.

[2] O. Baysal, M. W. Godfrey, and R. Cohen, "A bug you like: A framework for automated assignment of bugs," in *17th International Conference on Program Comprehension (ICPC)*. IEEE, 2009, pp. 297–298.

[3] S. Banitaan and M. Alenezi, "Tram: An approach for assigning bug reports using their metadata," in *3rd International Conference on Communications and Information Technology (ICCIT)*. IEEE, 2013, pp. 215–219.

[4] M. Linares-Vásquez, K. Hossen, H. Dang, H. Kagdi, M. Gethers, and D. Poshyvanyk, "Triaging incoming change requests: Bug or commit history, or code authorship?" in *28th International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 451–460.

[5] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani, "A time-based approach to automatic bug report assignment," *Journal of Systems and Software*, vol. 102, pp. 109–122, 2015.

[6] V. B. Sawant and N. V. Alone, "A survey on various techniques for bug triage," 2015.

[7] D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning bug reports using a vocabulary-based expertise model of developers," in *6th International Working Conference on Mining Software Repositories (MSR)*. IEEE, 2009, pp. 131–140.

[8] P. Bhattacharya and I. Neamtiu, "Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging," in *International Conference on Software Maintenance (ICSM)*. IEEE, 2010, pp. 1–10.

[9] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2009, pp. 111–120.

[10] L. Chen, X. Wang, and C. Liu, "An approach to improving bug assignment with bug tossing graphs and bug similarities," *Journal of Software*, vol. 6, no. 3, pp. 421–427, 2011.

[11] J.-w. Park, M.-W. Lee, J. Kim, S.-w. Hwang, and S. Kim, "Cost-aware triage ranking algorithms for bug reporting systems," *Knowledge and Information Systems*, vol. 48, no. 3, pp. 679–705, 2016.

[12] K. Aggarwal, T. Rutgers, F. Timbers, A. Hindle, R. Greiner, and E. Stroulia, "Detecting duplicate bug reports with software engineering domain knowledge," in *22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 211–220.

[13] W. Zou, Y. Hu, J. Xuan, and H. Jiang, "Towards training set reduction for bug triage," in *35th Annual Computer Software and Applications Conference*. IEEE, 2011, pp. 576–581.

[14] J. Xuan, H. Jiang, Y. Hu, Z. Ren, W. Zou, Z. Luo, and X. Wu, "Towards effective bug triage with software data reduction techniques," *IEEE transactions on knowledge and data engineering*, vol. 27, no. 1, pp. 264–280, 2015.

[15] (2016, Aug.) Jdt core component - eclipse. [Online]. Available: https://eclipse.org/jdt/core/