A Bug Assignment Approach Combining Expertise and Recency of Both Bug Fixing and Source Commits

Afrina Khatun and Kazi Sakib
Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh

Keywords: Bug Assignment, Bug Report, Commit Log, Tf-idf Weighting.

Abstract: Automatic bug reports assignment to fixers is an important activity for software quality assurance. Existing approaches consider either the bug fixing or source commit activities which may result in inactive or inexperienced developers suggestions. Considering only one of the information can not compensate another leading to reduced accuracy of developer suggestion. An approach named BSBA is proposed, which combines the expertise and recency of both bug fixing and source commit activities. Firstly, BSBA collects the source code and commit logs to construct an index, mapping the source entities with their commit time, which presents developers' source code activities. Secondly, it processes the fixed bug reports to build another index which connects the report keywords to the fixing time. Finally, on arrival of new reports, BSBA queries the two indexes, and combines the query results using tf-idf weighting technique to measure a BSBA score for the developers. The top scored developers are suggested as appropriate fixers. BSBA is applied on two open source projects - Eclipse JDT and SWT, and is compared with three existing techniques. The results show that BSBA obtains the actual fixer at Top 1 position in 45.67%, and 47.50% cases for Eclipse JDT and SWT respectively, which is higher than the existing approaches. It also shows that BSBA improves the accuracy of existing techniques on average by 3%-60%.

1 INTRODUCTION

Bug assignment is an essential step for fixing software bugs. When a new bug report arrives, an appropriate developer who is proficient in fixing that bug, needs to be identified from a huge number of contributing developers. This turns manual bug assignment into a difficult and time-consuming task. Any mistake in assigning the bugs to appropriate fixers may lead to unnecessary reassignments as well as prolonged bug fixing time. These factors raise the need of an automatic and accurate bug assignment technique capable of identifying potential fixers.

Automatic bug report assignment is generally performed using information sources such as bug reports, source code and commit logs (Anvik and Murphy, 2007). The fixed bug reports generally represent developer's activity in terms of fixing those. However, with the passage of time developers may move to different projects or company. As a result, considering only bug fixing activity may direct to suggestion of inactive developers. On the contrary, the source code and commit logs generally represent developers' activities in developing the system modules. However, a developer may not have fixed bugs related to all the modules. Analysing only source related activities may result in suggestion of inexperienced developers. So, considering only bug fixing or source related activities may suggest inactive or novice developers, which reduces the accuracy of bug assignment.

Understanding the importance of automatic bug assignment, various techniques have been proposed in the literature. In order to reduce the cost of manual assignment, Murphy et al. first proposed an approach based on text categorization of fixed reports (Murphy and Cubranic, 2004). BugFixer, another text categorization based method has been proposed by Hao et. al (Hu et al., 2014). This method constructs a Developer-Component-Bug (DCB) network, using past bug reports and suggests developers' list over this network. The list becomes less accurate with joining of developer or switching of development teams. Shokripour et. al (Shokripour et al., 2015) have proposed a time based approach that indexes source identifiers along with commit time to prioritize recent developers. However, the technique fails to achieve high accuracy due to considering only developers' source code activities. Tian et al. combines developers' pre-

Khatun, A. and Sakib, K.
A Bug Assignment Approach Combining Expertise and Recency of Both Bug Fixing and Source Commits.
In *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2018)*, pages 351-358
ISBN: 978-989-758-300-1

vious activities and suspicious program locations to identify bug fixers (Tian et al., 2016). However, this technique requires attached source patches with bug reports and also ignores the recent timing of source commits. An expertise and recency based bug assignment has been presented in (Khatun and Sakib, 2016). However, it ignored the bug report fixing recency of the developers. As a result, the developers are overprioritized with the experience of bug fixing.

A bug assignment technique, combining the expertise and recency of both bug fixing and source commit activities called BSBA (Bug fixing and Source commit activity based Bug Assignment) has been proposed. The overall bug assignment is performed using three steps - *Source Activity Collection*, *Fixing History Collection* and *Developer Suggestion*. The *Source Activity Collection* module takes source code and commit logs as input. It builds an index connecting source code identifiers with commits, to represent identifier usage owner and time. Besides, the *Fixing History Collection* module uses fixed bug reports to construct another index connecting bug report features (i.e. keywords) with the report fixer and fixing time. Finally, when new bug reports arrive, the *Developer Suggestion* module extracts the bug report keywords, and queries the indexes with these keywords. Based on the query results, a BSBA score is assigned to each developer using tf-idf weighting technique considering the experienced and recent use of keywords. The high scored developers are recommended as appropriate fixers.

BSBA, has been applied on Eclipse JDT and SWT for assessing it's compatibility. For these projects, the source code, commit logs and bug reports are collected from open source. BSBA's performance is measured using metric- Top N Rank [Shokripour et al., 2015]. In order to measure competency, it is compared with one source activity based technique known as ABA-time-tf-idf (Shokripour et al., 2015), one bug fixing activity based technique called TNBA (Shokripour et al., 2014) and one unified previous activity and program location based technique (Tian et al., 2016), which will be referred in remaining of this paper as Unified Model. The result shows that BSBA suggests 45.67% and 47.50% actual fixers at Top 1 position for Eclipse JDT and SWT respectively, which outperforms studied projects.

2 RELATED WORK

Concerned with the increased importance of automatic bug assignment, a number of techniques have been proposed by researchers. Significant works related to this research topic are outlined in following.

A survey has divided the existing bug assignment techniques into text categorization, tossing graph, source based techniques etc. (Sawant and Alone, 2015). Text categorization based techniques build a model that trains from past bug reports to predict correct rank of developers (Hu et al., 2014; Matter et al., 2009; Baysal et al., 2009). Baysal et al. have enhanced the text categorization techniques by adding user preference of fixing certain bugs in recommendation process (Baysal et al., 2009). 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 source related activities of developers, and may recommend developers who are either working in another project or company. As a result, inactive developers may get recommended which reduces prediction accuracy.

Tossing graph based bug triaging techniques for reducing reassignment have also been developed (e.g. (Bhattacharya and Neamtiu, 2010), (Jeong et al., 2009)). 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 (Jeong et al., 2009). Due to considering previous bug report information, the technique results in low accuracy of recommended list and search failure in case of new developer arrival.

Matter et al. have suggested Develect, a source based expertise model for recommending developers (Matter et al., 2009). 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 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 in recommendation process, which leads to assignment of bugs to novice or inexperienced developers. Subsequently, it increases bug reassignments, prolongs fixing time and reduces recommendation accuracy.

Time based bug assignment techniques have also been proposed in the literature. Shokripour et al. has

devised a technique focusing on using time meta data, while identifying the recent developers for bug assignment (Shokripour et al., 2015). The technique first parses all the source code entities (such as name of class, method, attributes etc.) and connects those 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 meta data. 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 expert fixers. Another Time aware Noun based Bug Assignment (TNBA) technique has also been proposed in (Shokripour et al., 2014). The technique first takes the fixed bug reports as input and identifies the fixers of the report. The fixers of the reports are determined using source patches or commit history. However, this technique ignores the source code activities of developers leading to suggestion of inactive developers.

Various techniques for automatic fixer recommendation have been proposed in the literature. Most of the techniques learn information from previous fix history or current commit history of software repositories. However, these techniques suffer from low accuracy due to recommendation of inactive or inexperienced developers. So, consideration of only one information source while assigning bug reports, cannot suggest accurate fixers.

3 METHODOLOGY

In this paper, a technique named Bug fixing and Source commit based Bug Assignment (BSBA) has been proposed, which combines developers' expertise and recency of both fixing and source related activities for accurate fixer suggestion. BSBA performs developer suggestion using three modules, which are - *Source Activity Collection*, *Fixing History Collection* and *Developer Suggestion*. The overall fixer suggestion procedure of BSBA is described below.

Source commits of developers generally reveal their source related activity information. Firstly, the *Source Activity Collection* module receives software source repository and commit logs as shown in Figure 1. It then extracts the source code entities from the source files and constructs an index which maps each source entity against the developers who committed the entities. This index represents the source code activities of the developers which will be later used by the *Developer Suggestion* module.

Next, *Fixing History Collection* module receives XML formatted bug reports as input as shown in Figure 1. The fixed bug reports indicate developers' bug solving history. So, this module extracts keywords from the bug reports and generates another index which associates the keywords to the developers, who previously fixed those keyword related bugs.

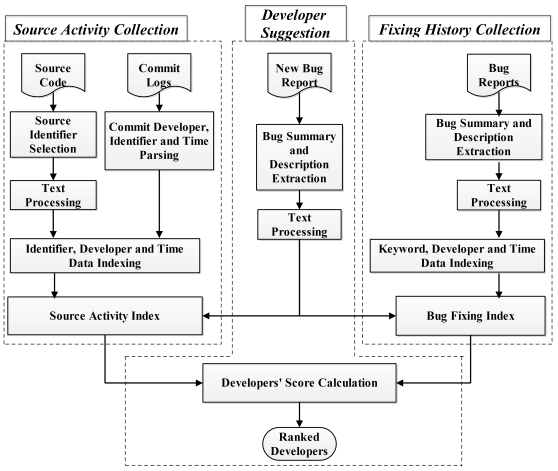

Figure 1: Overview of the ERBA.

Finally, *Developer Suggestion* module suggests an accurate developers' list on arrival of new bug reports. When a bug report arrives at the system, its keywords are extracted to formulate a query and searched in the constructed indexes. Lastly, tf-idf technique is applied on the search results to calculate a score for suggesting developers. In the following these modules are described in detail.

3.1 Source Activity Collection

The source code commits represent developers' expertise and recency of developing the system ((Matter et al., 2009)). The *Source Activity Collection* module is responsible for identifying developers' source activities. This module takes software source code and commit logs as inputs. It then parses the source files in the repository to extract the source entities. The names of classes, methods, attributes and method parameters are extracted as source entities. These data are taken into consideration as these entities represent developer vocabularies and activities.

Moreover, data by nature may contain unnecessary characters and keywords. For improved tokenization, all these parsed data undergoes through text pre-processing. The text pre-processing step contains identifier decomposition based on CamelCase letter and separator character, number and special character removal, stop word removal and stemming. In order to retrieve relevant results, both the complete and decomposed identifiers are considered for indexing.

The commit history of the project is also extracted

in XML format using git commands. A sample of the extracted commit history of Eclipse JDT is illustrated in Figure 2. The commit XML contains a number of attributes such as commit id (hash), author, commit date, subject and a list of changed identifiers associated with the commit. For example, the commit of Figure 2 shows that developer "Markus Keller" has committed changes to four source identifiers such as "FactoryPluginManager", "JavaElementLabelsTest", "JavadocView" and "JavadocHover" on *5 Dec, 2011*.

In order to determine developers' expertise and 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 phases of time during the project development. For searching recent usage of the identifiers by the developers, an index of all the identifiers needs to be built. So, Algorithm 1 is proposed which performs the task of developers' source activity collection. Algorithm 1 takes a

```
<commit>
<repo>eclipse.jdt.ui</repo>
<hash>b441737900fbbd3fe2346d6c05daa02ec061f6c1</hash>
<author>Markus Keller</author>
<authorMail>mkeller</authorMail>
<date>Mon, 5 Dec 2011 16:46:53 +0100</date>
<subject>Bug 357325: [render] Method parameter annotations
<note>Markus Keller</note>
<changedFiles>
    <identifier>FactoryPluginManager</identifier>
    <identifier>JavaElementLabelsTest</identifier>
    <identifier>JavadocView</identifier>
    <identifier>JavadocHover</identifier>
</changedFiles>
</commit>
```

Figure 2: Partial Commit Log of Eclipse JDT.

processed list of source identifiers as input. In order to associate the list of identifiers with corresponding developer list, a complex data structure, *SourceActivity* is constructed. It contains two properties - developer *name* and commit *date*. To construct the index, an empty *postingList* is declared (Algorithm 1, line 2). An empty variable *d*, of type *SourceActivity* is also declared in line 3 for populating the *postingList*. 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 changed, 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 is also defined to iterate on the *Commits* list (Algorithm 1, line 6). Each iteration of this loop initializes *d* with a new *SourceActivity* instance. It then updates the *name* and *date* property of *d* with *author* and *date* property of commit *c*, respectively (Algorithm 1, lines 7-9). For adding

Algorithm 1: Source Activity Collection.

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

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

identifiers into the *postingList*, the list is first checked whether it already contains the identifier (Algorithm 1 line 10-11). In the next step, the updated value of *d* is added to the *postingList* against the identifier (Algorithm 1, line 12). This *postingList* will be searched later using identifiers found in the arrived bug reports. Finally, Algorithm 1 returns an index of triplet data structure containing identifiers, developers who use those identifiers and their identifier using date.

3.2 Fixing History Collection

The fixed bug reports of a system is another key indicator of developers' proficiency of fixing similar bugs. The higher number of times and the more recent a developer fixed certain keyword related bugs, the higher potential the developer has to solve those keyword related bugs. The *Fixing History Collection* module performs the task of identifying expert and recent developers, who are proficient for solving similar bugs. A fixed bug report contains the full resolution history of the bug report. For this purpose, the bug reports are collected from bug tracking system, Bugzilla. The module then takes those bug reports for identifying developers' experience information. These bug reports are input in XML format for making it program readable. A partial XML structure of Eclipse bug #264606 is shown in Figure 3. It shows that the bug report contains a number of attributes such as *id*, *developer*, *creation_time*, *summary*, *description* etc. The more familiar a developer is with the keywords

```
<bug>
<id>264606</id>
<developer>Thomas ten Cate</developer><developer_username>ttencate
<creation_time>2009-02-11 17:30:00 -0500</creation_time>
<product>JDT</product>
<component>UI</component>
<resolution>fixed</resolution>
<bug_severity>normal</bug_severity>
<summary>[extract method] extracting return value results
in compile error</summary>
<description>Build ID: I20080617-2000 Use Extract Method on 'foo' :
    class Bar {
        boolean foo;
```

Figure 3: Partial XML Formatted Bug Report of JDT.

of fixed bug reports, the more expertise the developer has in fixing these keyword related bugs. As, a keyword related bugs can be fixed by several developers, an index is required which connects the keywords with the list of developers. For this purpose, another algorithm similar to Algorithm Algorithm 1 is used. It receives a list of fixed bug reports. Each report has three properties - a list of bug report keywords, the name of a developer who fixed the bug and bug fixing date. The keywords are collected from bug report summary and description. These keywords are then connected with the fixer and fixing date of the bug reports. As a result, another index containing the expertise information of the developers is constructed. This built index is later used by the *Developer Suggestion* module for ultimate fixer determination.

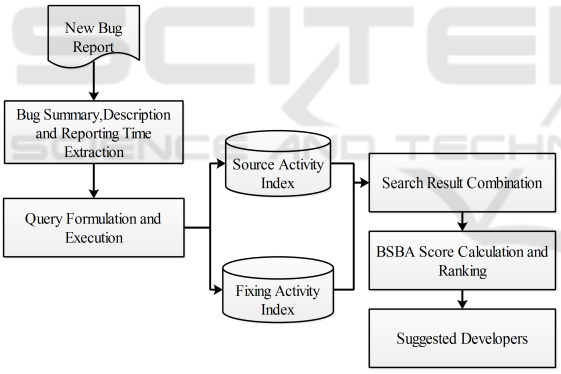

Figure 4: Developer Suggestion Procedure of BSBA.

3.3 Developer Suggestion

Finally, BSBA combines both the recency and expertise information gained from above mentioned indexes for appropriate developer ranking. The overall developer suggestion procedure is represented in Figure 4. When a new bug report arrives, *Developer Suggestion* module first extracts the *summary*, *description* and *reporting date* of the report. A search query is formulated using the keywords of the *summary* and *description* field of the report as shown in Figure 4. This query is then executed on the two indexes built by the previous two 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*. It represents a data structure consisting of four properties - *useFreq*, *fixFreq*, *useDate* and *fixDate*. The explanation of these properties is given in Table 1. Next, the

Table 1: Explanation of Attributes and Methods of Algorithm 2.

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

combined search results are used to calculate an expertise and recency score for each developer. This score is called a BSBA score. The overall suggestion is performed using Algorithm 2. The *getTermInformation* function takes bug report (*B*) as input, and performs the above mentioned search query formulation and execution task (Algorithm 2, line 2). Therefore, it returns a 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 *usage* and *fixation* are declared for storing each developer's source code and bug fixing information *d*, (line 5). The variables *useRecency* and *fixRecency* are initialized at line 6, for getting the time information associated with the source usage and bug fixation of the terms respectively. Next, an inner loop is constructed to iterate over the new bug report terms used by this developer (line 7). For each *term*, its corresponding information *t* of type *TermInfo* is extracted from *devTermInfo* (line 8).

For accurate developer recommendation, both the frequent and recent activities in the source code and bug fixing are important. tf-idf weighting technique is applied for measuring the frequent use of a term in

Algorithm 2: Developer Suggestion.

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

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

the source code by a developer (line 9). This technique determines the weight of a term using the frequency of its usage ($useFreq$) and the generality of it in the project ($log(\frac{\#dev}{devUseFreq(t)})$). Considering only frequent use of terms in source code may result in recommendation of inactive developers. To alleviate this problem, the recent use of a term is incorporated with its frequent use in the source code. The recency of a term is determined by adding the inverse of the number of developers who used this term, and the inverse of date difference between the bug reporting date ($Date$) and this term using date ($useDate$) (line 10). The greater the value of *devUseFreq(t)* is, the more important the term is for the developer. Again the greater the interval ($B.Date - t.Date$) is, the less recent the term is used. This recency of a term usage ($useRecency$) is then multiplied by its frequency ($tfIdf$) to calculate developer's source usage information on this term. Finally, the sum of the usage of all the new report's terms determines the developer's source code activities regarding these terms (line 11).

On the other hand, the expertise and recency of a developer, in resolving a term is calculated using similar tf-idf weighting technique. It determines the experience of fixing a term, by multiplying the term fixing frequency (*fixFreq*) with the generality of fixing the term related bugs ($log(\frac{\#dev}{devFixFreq(t)})$) (line 12). The developer who fixed a term recently should get higher priority than the developer who fixed it a long time ago. Hence, the recent fixation of a term is incorporated with the expertise weight in a similar manner as shown in line 13. Developer's bug fixing expertise and recency is obtained by summing the fixing expertise and recency of all terms (line 14). Lastly, for each iteration of *devTermInfo*, an instance *dev*, of type *Developer* is constructed, initialized (using developer name and BSBA score) and inserted into *devList* (line 15-18). The technique incorporates recency and expertise values for assigning BSBA score to developers. As a result, Algorithm 2 ends its task by sorting the *devList* in a descending order based on BSBA scores. BSBA concludes its task by suggesting the developers at the top of the list as fixers for handling the recently arrived bugs.

4 RESULT ANALYSIS

For evaluating the compatibility of BSBA, experimental analysis have been conducted on two projects - Eclipse JDT and SWT. The experimental analysis is focused to evaluate the ranking of the actual fixers in the suggested list. The suggested list is evaluated using metric, Top N Rank. The results of BSBA is compared with ABA-time-tf-idf (Shokripour et al., 2015), TNBA (Shokripour et al., 2014) and Unified model (Tian et al., 2016). The details of the experimental arrangements are demonstrated below.

4.1 Dataset

Table 2 enlists the properties of the two studied projects. These projects are also used in evaluating bug assignment techniques (Shokripour et al., 2015), (Tian et al., 2016). The collected experimental data are available in the repository of BSBA (bsb, 2017). Top N rank or accuracy refers whether the top N suggested developers contain the actual bug fixer, where

Table 2: Properties of Experimetal Dataset.

Project	No. of Commits	No. of Classes	No. of Bug Reports
JDT	25,700	6,899	6,274
SWT	24,265	1,981	4,151

N=1,3,5,...,n (Shokripour et al., 2015). If the top N developers contain any of the actual fixers, the suggestion is considered correct. For example, N=3 refers, an actual fixer is obtained in the top 3 suggested developers. A higher value of this metric represents higher accuracy of BSBA.

Table 3: Performance of BSBA on Two Studied Projects.

Proj -ect	No. of Test Report	Top 1	Top 3	Top 5	Top 10
JDT	600	274 45.67%	466 80.00%	522 87.00%	584 97.33%
SWT	400	190 47.50%	311 77.75%	354 88.50%	399 99.75%

Table 3 shows the Top N ranking performance of BSBA. The number of times actual fixers obtained within the Top 1, Top 3, Top 5 and Top 10 positions are calculated. For example, in case of Eclipse JDT, the recent 600 reports (out of 6274) sorted by date, are selected for testing purpose. It also shows that out of 600 test reports, 274 times (i.e. 45.67%) BSBA shows the correct developer at position 1. That is, in 45.67% cases the first developer may correctly fix the bug without reassigning it to other developers. Moreover, 80.00%, 87.00% and 97.33% cases the actual fixers are obtained at the Top 3, Top 5 and Top 10 ranks respectively. These, higher values indicate that BSBA can successfully place the actual fixers at the top of the suggested list. In case of SWT 47.50% actual fixers are obtained at position 1, which is also promising. It also depicts that for all the three projects above 97.00% of the fixers can be suggested by BSBA.

In order to identify whether the actual fixers are suggested at the top of the list by BSBA, the number of fixers obtained at each *Nth* rank position is plotted in Figure 5. It shows that with the increasing size of the rank position, the number of obtained developers at that rank position is decreased. For both projects, the ranking of the developers has followed a pattern near similar to positive skewness. This skewness refers that most of the developers are obtained at the left side of the graph, which indicates that BSBA suggests most of the actual developers at lower rank positions that is at the top of the lists. Besides, it is also visible from Figure 5 that, for each of the three projects, the highest number of developers are achieved at Top 1 position.

Figure 6 and 7 show the comparison of Top N Ranking accuracy among ABA-time-tf-idf (Shokripour et al., 2015), TNBA (Shokripour et al., 2014), Unified Model (Tian et al., 2016) and BSBA.

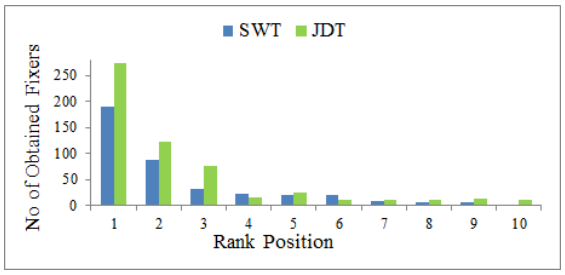

Figure 5: Top N Ranking of BSBA on Two Projects.

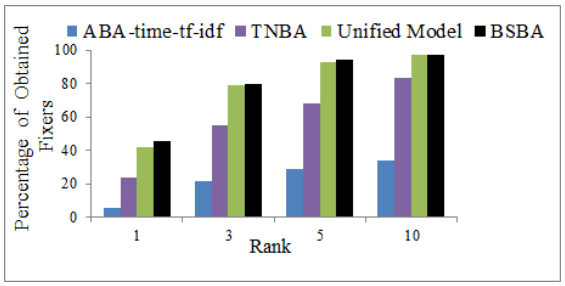

Figure 6: Comparison of Top N Ranking on Eclipse JDT among ABA-time-tf-idf, TNBA, Unified Model and BSBA.

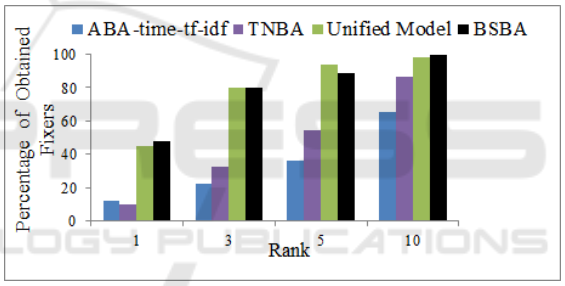

Figure 7: Comparison of Top N Ranking on SWT among ABA-time-tf-idf, TNBA, Unified Model and BSBA.

For example, Figure 6 shows that for each of the positions, BSBA achieves higher bar than the existing ones in JDT. It retrieves 45.67% fixers within position 1 which is higher than ABAtime-tf-idf (5.85%), TNBA (23.83%) and Unified Model(42%). A similar trend is also found while Top N ranking in SWT (Figure 7). Besides, the reason behind BSBA's bar at position 5 for SWT, being lower than Unified Model is that, BSBA shows most of the developers before position 5. For each of the studied subjects, BSBA outperforms ABAtime-tf-idf, TNBA and Unified Model while ranking the developers. These figures also show that BSBA outperforms the accuracy of existing techniques on average from 3% ((Tian et al., 2016)) upto 60% ((Shokripour et al., 2015)). These improvement of ranking ensure higher accuracy of BSBA due to combining both recency and expertise information which is ignored by the existing approaches.

5 THREATS TO VALIDITY

This section presents the threats which can affect the implementation and evaluation validity of BSBA. The naming convention of the source entities, the quality of bug reports and commit logs can introduce redundant information, which can reduce the accuracy of assignment. However, this risk is also minimized by collecting the source repository, commits, and bug reports from widely used information tracking systems such as git and Bugzilla. Analysing the performance of BSBA with other metrics can also affect the generalization of the results. This risk is handled by selecting metric which is widely applied in evaluation of bug assignment techniques.

6 CONCLUSION

The paper proposes a bug assignment technique named as BSBA which considers the expertise and recency of both bug fixing and source committing of developers . The *Source Activity Collection* module firstly takes source code and commit logs as input, and constructs an index for associating the source entities with the commit time of entities. In order to collect the bug fixing activity of developers, *Fixing History Collection* module builds another index which connects the bug report keywords with fixing time. Lastly, *Developer Suggestion* module queries the two indexes with new reports' keywords, and applies tf-idf weighting on the query result to calculate a BSBA score for all developers. Actual fixers are selected based on the highest scoring developers.

The applicability of BSBA is experimented on two projects - Eclipse JDT and SWT. The experimental results are compared with existing approaches. For Eclipse JDT, BSBA shows 45.67% of the actual fixers at Top 1 position which is higher than the existing three approaches (5.85%, 23.83%and 42%). For SWT, in 47.50% cases actual fixers are obtained at position 1 respectively.

In future, the bug reports which keywords are not found in the bug and source activity indexes, needs to be considered. Besides, handling newly joined developers, who owes no bug fixes or source commits, can be another future direction.

REFERENCES

(2017). Afrina/Expertise And Recency Based Bug Assignment. URL: https://drive.google.com/drive/folders/0B_Jc3FdEOCHzakxqRFg0ckVxOXM?usp=sharing [accessed: 2017-03-05].

Anvik, J. and Murphy, G. C. (2007). Determining implementation expertise from bug reports. In *4th International Workshop on Mining Software Repositories (MSR)*, page 2. IEEE Computer Society.

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

Bhattacharya, P. and Neamtiu, I. (2010). Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In *26th International Conference on Software Maintenance (ICSM)*, pages 1–10. IEEE.

Hu, H., Zhang, H., Xuan, J., and Sun, W. (2014). Effective bug triage based on historical bug-fix information. In *25th International Symposium on Software Reliability Engineering (ISSRE)*, pages 122–132. IEEE.

Jeong, G., Kim, S., and Zimmermann, T. (2009). Improving bug triage with bug tossing graphs. In *7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE)*, pages 111–120. ACM.

Khatun, A. and Sakib, K. (2016). A bug assignment technique based on bug fixing expertise and source commit recency of developers. In *19th International Conference on Computer and Information Technology (ICCIT)*, pages 592–597. IEEE.

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

Murphy, G. and Cubranic, D. (2004). Automatic bug triage using text categorization. In *16th International Conference on Software Engineering & Knowledge Engineering(SEKE)*, pages 1–6.

Sawant, V. B. and Alone, N. V. (2015). A survey on various techniques for bug triage. *International Research Journal of Engineering and Technology*, 2:917–920.

Shokripour, R., Anvik, J., Kasirun, Z. M., and Zamani, S. (2014). Improving automatic bug assignment using time-metadata in term-weighting. *IET Software*, 8(6):269–278.

Shokripour, R., Anvik, J., Kasirun, Z. M., and Zamani, S. (2015). A time-based approach to automatic bug report assignment. *Journal of Systems and Software*, 102:109–122.

Tian, Y., Wijedasa, D., Lo, D., and Le Gouesy, C. (2016). Learning to rank for bug report assignee recommendation. In *24th International Conference on Program Comprehension (ICPC)*, pages 1–10. IEEE.