Why So Complicated? Simple Term Filtering and Weighting for Location-Based Bug Report Assignment Recommendation

Ramin Shokripour*, John Anvik†, Zarinah M. Kasirun*, Sima Zamani*

*Faculty of Computer Science & Information Technology
University of Malaya, Kuala Lumpur, MALAYSIA
shokripour@siswa.um.edu.my
zarinahmk@um.edu.my
sima.zamani@siswa.um.edu.my

†Department of Computer Science
Central Washington University, Ellensburg, Washington, USA
janvik@cwu.edu

***Abstract*—Large software development projects receive many bug reports and each of these reports needs to be triaged. An important step in the triage process is the assignment of the report to a developer. Most previous efforts towards improving bug report assignment have focused on using an activity-based approach. We address some of the limitations of activity-based approaches by proposing a two-phased location-based approach where bug report assignment recommendations are based on the predicted location of the bug. The proposed approach utilizes a noun extraction process on several information sources to determine bug location information and a simple term weighting scheme to provide a bug report assignment recommendation. We found that by using a location-based approach, we achieved an accuracy of 89.41% and 59.76% when recommending five developers for the Eclipse and Mozilla projects, respectively.**

***Index Terms*—Bug Report Assignment, File Activity Histories, Named Entity Recognition, POS Filtering, Mining Software Artifacts.**

I. Introduction

Software projects commonly use issue tracking systems (ITS) such as Bugzilla[1] and Jira[2] as a means for accessing and organizing change requests, issues reports, or bug reports.[3] Bug reports provide a way for the software project to manage the identification of faults or requests for new features.

However, the use of an issue tracking system by a software project is not without a cost. Large software development projects such as Mozilla[4] and Eclipse[5] receive many new reports daily. For example, as of October 2012, the Mozilla project had received over 800,000 reports, averaging 300 new bug reports each day. Each of these new bug reports requires *triaging*. A project member, called a *triager*, must examine each recently submitted bug report and makes decisions about how the report will be organized within the development process of the software project. Such decisions include the validity of the reported fault, such as if the fault has been previously reported (i.e. a duplicate report) or if the cause of the fault is that of third-party software. These triage decisions also include if the bug report has been filed against the correct product component or has been given an appropriate priority. A key decision made by the triager is who will be assigned to make the code changes necessary to fix the fault or implement the new feature. This assignment decision can have important consequences for the project, as an incorrect decision can increase the time taken for fixing a bug [1] and therefore increase the cost of the project [2].

A project's triage process can consume a significant amount of time and resources [1]. Often the triager is also a member of the development team. Time spent triaging bug reports is time not spent improving the software product, and therefore represents an overhead to the project. Any reduction in time spent on the triage process frees resources for software product improvement. Also, the speed at which bugs reports are triaged, especially those resulting in a code change, can have an effect on the perceived quality of the software project. This is particularly true for projects that use a Open Source Software (OSS) development process where the responsiveness of the developers to the project's user community is partially measured by how quickly bugs are fixed [3].

[1]http://www.bugzilla.org/

[2]http://www.atlassian.com/software/jira

[3]Although change requests commonly contain information about both software faults and feature requests, we will use the colloquial term "bug report" to refer to both types of reports.

[4]http://www.mozilla.org

[5]http://www.eclipse.org

There have been many efforts towards reducing the cost of the triage process. These efforts include assessing the quality of bugs [4], automatically identifying duplicate bug reports [5], [6] and providing recommendations for the assignment of bug reports [7], [8], [9], [10], [11], [12]. Prior work on bug assignment recommendation has focused primarily on the use of either machine learning or information retrieval techniques to predict the developer most suited to resolve a bug. These systems commonly determine the expertise of the project developers based on reflections of their activities within project artifacts. We refer to such approaches as *activity-based*.

Although activity-based approaches to bug report recommendation have been found to be highly accurate [2], [4], [12], they are not without weaknesses (see Section II). An alternate to an activity-based approach is to recommend a bug report assignment based on the predicted location(s) for the bug in the source code. We refer to these approaches as *location-based*. Such approaches are similar to those for impact analysis [13], [14], but differ in intent.

This paper presents a two-phased location-based approach for bug report assignment. In the first phase we predict the source code files that will be changed to fix a new bug report. Specifically, we determine the parts-of-speech (POS) for terms from text information sources to create an index of unigram noun terms that link to source code files. This results in a simpler term index than used in similar approaches. Also, unlike prior approaches that limit themselves to a single source of information (i.e bug report descriptions or source code revision commit messages), our approach uses four distinct information sources to populate the term index.

In the second phase of the approach, we use the predicted source code files from the first phase to recommend developers for handling the new bug report. We show that our approach avoids the complex computations common with activity-based approaches through the use of a simple method for term weighting based not on the frequency of terms within information sources, but the term frequency across multiple information sources. We applied our approach to two software projects, Eclipse and Firefox, and achieved an accuracy of 89.41% and 59.76%, respectively. Finally, we also show that the approach results in a higher accuracy when compared to another location-based approach.

The rest of this paper is organized as follows. First, we present motivation for our location-based approach to bug report assignment recommendation. Next, we present our location-based approach in Section III and evaluate the accuracy of the approach in Section IV. Finally, we discuss threats to the validity of our work in Section V and related work in Section VI, before concluding the paper.

II. Motivation

In this section, we present the motivation behind our location-based approach to bug report assignment recommendation. First, we discuss some of the drawbacks of activity-based approaches compared to location-based approaches. Then we discuss the use of different information sources for making an assignment recommendation. Finally, we discuss the types of extracted entities that are used in either activity-based or location-based approaches.

A. Activity-Based vs. Location-Based Approaches

As previously mentioned, bug report assignment recommendation approaches can be divided into two categories: activity-based and location-based. In an activity-based approach, an assignment recommendation is made based on developer expertise information gleaned from reflections of activity in the project's artifacts, particularly bug reports.

Alternatively, an assignment recommendation can be made based on which source code files will need to be changed, or the location of the bug within the project, and which developers work at those locations.

In activity-based approaches, information resources are mined to extract information about various developer activities. However, such an approach can fail in the following ways:

1) *New developers.* If the developer that resolves a bug report is new to the project, there will be a period of time in which this developer will not be recommended by an activity-based approach. This is due to the developer not having generated enough activity information to be recommended by such an approach.
2) *Developers switching teams.* Software projects commonly organize developers into teams. However, after a period of time a developer may move from one project team to another. In such a case, an activity-based assignment approach will continue to recommend that the developer fix bugs related to their former project area until their activities generate enough information to make a correct recommendation. Depending on the amount of activity information the developer previously generated as a member of that team and the sensitivity of the activity-based approach to recent developer activity, it may be some time before the recommender makes a correct recommendation for the developer.
3) *Reliability of activity information.* Sources of developer activity information are commonly noisy. For example, the "assigned-to" field for bug reports of several projects was found to not reflect the developer who actually resolved a bug, but another project member [8]. Such noise can lead to activity-based approaches making incorrect recommendations.

Location-based bug assignment methods avoid these problems by providing a better representation of developers' current expertise. Activity-based approaches commonly have to use data from over a long period of time in order to have sufficient project expertise information to make a recommendation. This increases the likelihood that an activity-based approach will make a recommendation based on obsolete expertise information. In contrast, developers that have recently fixed a fault in a source code file are more likely to have the necessary expertise to fix a new bug in the same location than other project developers [15]. If a new developer joins a project team or a project member changes teams, their

contributions will immediately be recognized by a location-based approach. This means that the developer is likely to be recommended correctly for bug report assignment sooner than with an activity-based approach, resulting in a higher quality of assignment recommendation.

Another advantage of a location-based bug report assignment recommender is a bounding of recommendations. It is uncommon for multiple developers to work in the same location of the source code. On average only two developers will work in the same area of a source code file [16]. This leads to a smaller upper bound on the number of possible recommendations than is typical for activity-based approaches.

B. Information Sources

An important consideration in the accuracy of a bug report assignment recommender is the information source that is used for determining developer expertise. As a developer improves a software product he leaves evidence of his expertise in various software artifacts. In spite of the diversity of data that is available from a software project, bug report assignment recommendation approaches tend to focus on only one source of expertise information, commonly either bug reports [12], [17] or source revision commits [11]. If the data source is limited or not available, there may not be sufficient information for the use of a particular approach or the approach could have a low accuracy. For example, if an approach exclusively uses bug report information, it may not be effective for a newly created software product that has generated very few bug reports. Also, by using a diverse set of information sources, noise caused by dominant cross-cutting terms in a single information source is reduced. We therefore focused on an approach that used multiple project artifacts for making assignment recommendations. Specifically, we focused on using information from source code repository commits, identifiers and comments in the source code, and information from previously fixed bugs.

C. Extracted Entities

The entities that are used for establishing the relationship between a new bug and artifacts of the project has a significant role on performance of the approach. Most proposed approaches for automatic bug assignment recommendations remove noise by only using general preprocessing steps of the natural language processing (NLP), such as removing stop words and non-alphabetic tokens. However, Capobianco et al. [18] showed that using only unigram noun terms significantly improves the accuracy of IR-based traceability recovery method. Unlike other parts of speech, such as verbs and adjectives, nouns are usually used in a specific context. When two people use the same noun, their purposes are often related. Moreover, using only nouns improves the accuracy of our approach by avoiding the addition of noise words into our index [19]. Therefore, we focused on the use of only noun entities for terms in our proposed approach.

III. Proposed Method

In this section we present our location-based approach for automatic bug report assignment. There are two phases to our approach. First, we predict the source code files that will be changed to resolve a new bug report. Next, we recommend the developers for the new report based on information about who has previously fixed faults in the predicted source code files. Note that the accuracy of the first phase has a large influence on the results of the second phase and therefore the overall accuracy of the approach.

As mentioned in Section II, our approach focuses on using only unigram noun terms. To predict the source code files that will be changed to fix a new bug, we create an index of nouns where each unigram noun is linked to one or more source code file. To populate our noun index, we extract the nouns from the textual information of four different artifacts produced during software development: identifiers in the source code, commit messages from the version control system (VCS), source code comments, and the summary and description of bug reports marked as FIXED in the issue tracking system. Fig. 1 provides an overview of our approach.

The rest of this section proceeds as follows. First, we describe the four information sources that we use to populate our noun index. Next, we explain our process for extracting nouns. Then we describe our technique for weighting the terms used for predicting the location of a bug in the source code (Section III-C) before describing how we map source code files to project developers (Section III-D). Finally, we describe how we make a bug report assignment recommendation using the two-phased location-based approach in Section III-E.

A. Information Sources

1) Identifiers: Identifiers are the names of classes, methods, fields and parameters in the source code. Identifiers play a significant role in extracting information from the source code because developers select identifiers very carefully [20]. For example, in the "accessibility" package of a project, the word 'access' is usually combined with other words to create the identifiers for classes and methods in the package. That is to say, the words selected for identifiers have a meaningful relationship with that part of the source code. Therefore, words used to create identifiers can help to determine the area of responsibility for each file in the project.

2) Commit Messages: The messages that developers write when committing their changes to the VCS typically contain information about their development activities in the different source code files. Even if some developers do not provide a message when committing their changes or the message is short, the combination of all of the commit messages for a source file provides information about the usage and responsibility of the file in the project. In the commit messages, developers tend to include the reason for changes, such as adding a new feature or fixing a bug. Also, with each commit the path of the changed file is stored. This information helps in correlating the commit messages to their corresponding files.

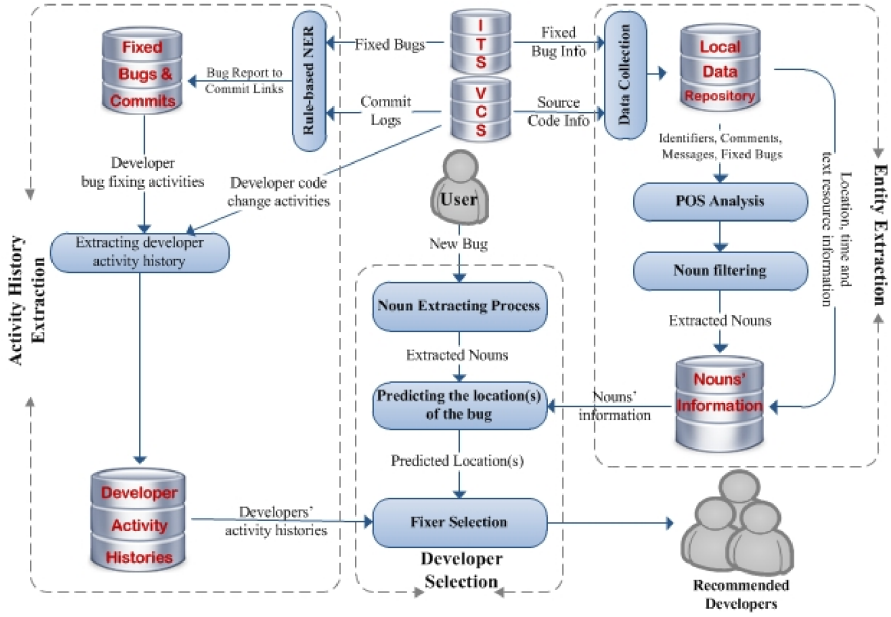

Fig. 1. An overview of the approach.

3) Comments in Source Code: Developers commonly provide short descriptions for lines of source code. These descriptions may contain such information as the reason for adding those lines to the code or why those lines were changed. In our work, we keep track of the source code files from which we extract comment information so as to establish the relationship between the terms and source code files.

4) Reports of Previously Fixed Bugs: Previously fixed bug reports provide an important source of information for our approach. However, we must first link the bug reports to source code files. The source code file(s) associated with a fixed bug report can be determined in a number of ways. First, some bug reports have attachments that are patches containing the files, or portions of the files, that are affected by a fix. Also, bug reports may have comments that refer to a specific commit to the project's VCS, or source commit messages may contain a bug report id. Finally, the nouns appearing in the bug report's summary or description can be used to link bug reports to source code files. For example, we found that for the Eclipse project, approximately 18% of the nouns used in the summaries and descriptions of new bug reports were similar to the nouns that occurred in either VCS commit messages, identifiers or source code comments.

B. Extracting Nouns from Project Artifacts

Recall that the first step of our approach is predicting which source code files will be affected by fixing a new bug. In this section we describe how we create our noun index.

First, we populate our index with nouns extracted from the identifiers found in the source code files. Depending on the programming language used by the project, we use a different software package. For extracting the identifiers from Java source code files, such as for the Eclipse project, we use Jeldoclet[6], a tool that can export the contents of Javadoc comments as XML. The identifiers that we extract are the class names, method names, fields and parameters. For a project that uses C++, such as Mozilla, we use CTAGS[7] to extract the same types of identifiers.

Identifiers are usually composed by the concatenation of a set of words. We decompose the identifiers using the approach recommended by Butler et al. [21] to produce a set of noun terms. However, we found that bug reporters will sometimes directly mention the name of the classes or methods in the description of the problem. Therefore we also include the class names and method names in our noun index. We then link the identifier nouns to their corresponding source code files.

Next, we add to our index the nouns found in the commit messages of the project's VCS. As the projects we investigated used CVS as their VCS, we use the CVSANALY[8] package to extract all source code commits, including their commit messages, to a database. The commit messages are then tokenized and filtered as described later in this section, and links made between the nouns and source code files.

To populate our index with nouns from the comments found in source code files, we use the scr2srcml package of srcML[9].

Finally, we extract the nouns found in bug reports, specifically those reports that have been marked as FIXED in the issue tracking system. We first create a list of report ids for the reports with the status FIXED in the project's issue tracking system. We then download the corresponding bug reports and store them as XML.

[6]http://jeldoclet.sourceforge.net/
[7]http://ctags.sourceforge.net/ctags.html
[8]http://metricsgrimoire.github.com/CVSAnalY/
[9]http://www.sdml.info/projects/srcml/

```
3.3 maintenance - Fix for 101610
fix for #10214
Merged fixed for bug 102494 into R3__maintenance branch
fix for 10382 Super type hierarchy computed on selection change in Outline
Real fix for 10881
fixed 27490, 27491
```

Fig. 2. Examples of the term 'fix' and bug ids appearing in commit messages.

Having collected the bug reports, we next correlate the fixed bugs to source code files of the project. We use two techniques to link source code files to bug reports. First, we examine any patch(es) attached to the bug report. From these patches we extract the names of the changed files.

If the bug report does not have an attached patch, the commit messages in the VCS are used to determine the link. For the projects that we examined, we found that developers commonly put the bug report id in the commit message when submitting changed files to the project's VCS. We therefore use the bug report id(s) appearing in the commit messages to determine the link between a bug report and the source code files. This approach to linking bug reports and source code files has also been used by other researchers [2], [22].

Developers also use key words such as 'fix' and 'bug', in addition to bug report ids in commit messages. However, the use of these key words is ad-hoc. Fig. 2 shows some examples where the term 'fix' and a bug report id are used in various commit messages. We can see that there is no common format or convention followed.

To detect the bug report ids in various cases of using key words and ids in a commit message, we use a ruled-based Named Entity Recognition (NER) method [19], specifically the NE transducer component of the ANNIE[10] plugin of GATE[11]. The Named Entity transducer uses JAPE[12] grammars for defining rules that detect these entities. The rules can be defined based on the different results of the various steps of text analysis, such as a part-of-speech (POS) tagger, morphological analysis, or a combination of them.

To improve the confidence of the results from using NER, the number extracted from the commit messages is compared with the ids of the collected bug reports. If the number exists in the list, then the date of the source revision commit is compared with the creation and resolution date of the bug report. If the commit date is after the creation date of the report or before the resolution date of the report, then the committer (the person that committed the change to the project's VCS) and the fixer (the person who changed the status of bug to FIXED in the project's ITS) are compared. If the names match, then the CVS commit is linked to the bug report. As the username in the project's ITS may not be the same as the username in the project's VCS for the same developer, to compare the two names we created a map between the usernames in the FIXED bug reports which were used as an information resource and the usernames in the extracted commit messages of our data set.

Having established links between the collected bug reports and project's source code files, we then extract nouns from the bug reports. Textual information in bug reports is known to be noisy [4]. For example, sometimes in an effort to assist in the resolution of a bug, a reporter will add additional text to the bug report to help with the discussion of the problem or a proposed fix. However, this type of text adds noise that can weaken the accuracy of a bug report assignment approach. This noise appears primarily in the comments of the bug reports. In some of the projects, such as the JDT product of the Eclipse project, reporters often add sample code, stack traces, error logs and other types of the data to the bug report. We therefore restrict the textual data used in our approach to that from the summary and description of the bug report. The summary and description are tokenized and we add any new nouns to our index, as well as any newly found links between an existing index term and source code file(s).

Having extracted nouns from the four information sources, we further refine the index terms by removing terms that are less than three characters, are a symbol, or start with a digit.

The role of words in a sentence is important in determining the value of those words. As mentioned in Section II, we only use nouns in our index. When determining the nouns to add to the index from a text source, such as from a bug report description or a commit message, we use the ANNIE plugin of GATE. This plugin is used for sentence splitting, tokenizing and POS tagging of the terms. After analyzing the text, we retain all of the words in the POS categories of nouns (e.g., NN (noun - singular or mass) and NNP (proper noun)).

C. Weighting the Index Terms

For each noun-file pair in the index, we calculate a weight based on the number of information sources from which the noun came. We count the number of times a noun appears in a bug report, a commit message, a source comment or an identifier, and then record this number for each noun-file pair. For example, the word 'position' for the file "/org.eclipse.jdt.ui/ui/org/eclipse/jdt/internal/ui/text/correction/JavaCorrectionAssistant.java" appeared in three information sources: a commit message for the file, the identifiers in the file and the description of the fixed bug report linked to the file. Therefore, the weight for the term 'position' for this file is 3.

D. Determining Relevant Developers

The second part of our approach is to determine the set of developers to recommend for each source code file. We found that the developers who have fixed previous bugs of a file are more likely to fix new bugs in the file than other developers who have worked in the file. Therefore we only extract reports marked as FIXED when creating our noun index.

We create a second index that maps each source code file found in our first index to a developer. The names are

[10]http://www.aktors.org/technologies/annie/
[11]http://gate.ac.uk/
[12]Jolly And Pleasant Experience

TABLE I
WEIGHTS OF THE COMMON NOUNS BETWEEN THE ECLIPSE BUG 100233 AND THE FILE "/ORG.ECLIPSE.JDT.UI/UI/ORG/ECLIPSE/JDT/INTERNAL/UI/TEXT/ CORRECTION/ JAVACORRECTIONASSISTANT.JAVA

Nouns	Weight
Annotation	3
source	3
editor	2
error	2
hierarchy	2
marker	2
parameter	2
type	2
class	1
following	1
problem	1
string	1
Total	**22**

determined by the "Line-10 Rule"[13] [23], [24], [25] to be the set of developers that committed changes to the project's VCS for a particular source code file. We consider the developer who most recently fixed a bug in the predicted source code file to be the most appropriate developer for resolving a new bug report.

E. Making an Assignment Recommendation

To make a bug report assignment recommendation for a new bug report, first we predict the source files for the new bug report. To do this we extract and filter the nouns found in the report's summary and description as described in Section III-B. This results in a set of nouns.

We then look up each of these nouns in the noun index. This results in a set of predicted files for the new bug report. For each of these files we compute a relevance measure. Eq. 1 shows the formula used to calculate the file relevance. File relevance is defined as the sum of weights for each noun that is common between the bug report and a file. For example, Table I shows the weights of the common nouns between the Eclipse bug 100233 and the file "/org.eclipse.jdt.ui/ui/org/eclipse/jdt/internal/ui/text/correction/ JavaCorrectionAssistant.java". We see that the relevance value of this file is 22 (i.e. the sum of the weights).

Bug report assignment recommenders that use text information sources commonly use inter-/intra-document frequency (i.e. TF-IDF) to determine the relevance of terms. In our approach we focus on how the terms relate to describing the responsibility of a source code file within the project. We believe that if a term is important in describing the responsibilities of a file, then the term will appear across multiple information sources. Although there may be cases where this assumption is violated, we found in our work that the assumption holds well.

$$Relevance = \sum_{Common\ \ Nouns} Nouns_{weight} \quad (1)$$

[13]This rule refers to using a specific line of the commit message to determine the user name of the committer.

We further restrict the bug's predicted location to those files that are part of the project's component as indicated in the bug report. As the reports that we are using for evaluation are marked as FIXED, there is a high probability that the component information is correct.[14] Also, we observed that the average number of changed files for a fixed bug report is two. Therefore we recommend the two files with the highest relevance measure as the predicted locations for each bug.

It is impossible to determine if the predicted locations are the actual bug fix locations prior to fixing the bug. Therefore, we assume that the predicted locations are either the actual file(s) that will be changed to fix the bug, or files that are related to the actual file(s). Having predicted the source code files for the new bug report, we then look up each of the predicted source code files in our source code file index to determine the developer recommendation for each file. The developers are then ranked according to their expertise.

$$Expertise(d, f, r) = \sum_{All\ \ act(d,f)} \frac{1}{\sqrt{date_{act} - date_{create}}} \quad (2)$$

We calculate the developer's expertise for fixing the bug using Eq. 2. This equation defines the expertise of each developer (d) on a predicted file (f) for the bug report (r) based on the number of previous change activities (act) for the file by the developer, and the date difference (in days) between the report creation date ($date_{create}$) and the change activity date ($date_{act}$). In other words, we determine the expertise of the developers based on who has changed the predicted file(s) the most recently. In this way, a developer who has the most recent bug change activity for a predicted file is more likely to be recommended. For example, suppose that a bug report is created on Day 10 of the project. Developer A commits a fix for the bug on Day 15 and Developer B commits a revised fix on Day 20. Then the expertise of Developer A for the file will be 2.24 and the expertise of Developer B will be 3.16. Developer B will be recommended over Developer A.

IV. EVALUATION

To evaluate the effectiveness of our approach, we chose to use the measure of *accuracy*. We chose this measure for two reasons. First, accuracy measures the ability of our approach to correctly recommend the developer that actually fixed the bug, which is one of our questions in conducting this research. Second, we wanted to compare our results to those of other approaches, and those works also used accuracy as their evaluation metric (see Section IV-C). Eq.(3) shows how we measured the accuracy of our approach.

$$Accuracy = \frac{\sum_{\#\ of\ \ reports}^{i=\ 1} 1\ if\ \ correct,\ 0\ otherwise}{\#\ of\ \ reports} \quad (3)$$

We evaluated our approach using two popular open source projects: Eclipse and Mozilla. So as to work with a manageable set of data, we restricted our data to the Debug component

[14]In practice this filtering may result in an incorrect recommendation for a new bug report if it is miscategorized.

TABLE II
DATA SET USED FOR EVALUATION.

	# of commits	# of files	# of developers	Test Set size
JDT-Debug	7,696	702	9	85
Mozilla Firefox	1,623	47	57	80

of the Eclipse JDT project and the Mozilla Firefox project. We also restricted our data to the commits that were made to the JDT and Firefox projects before December 2006 and January 2007, respectively. We chose this time frame so that we could analyze the commit messages for each file of the project since the creation of the file. Table II shows the number of commits, files and developers for each of the projects.

To test the accuracy of our approach, we selected the last 100 bug reports from each of the two projects as our testing sets. The earliest bug reports in our test sets were reported on Feb 2, 2005 for JDT and Feb 14, 2006 for Firefox. We then examined these reports and determined the files that were changed for each fix and the developers that made these changes. We were not able to determine this information for all of the selected reports. As a result, our test set for the JDT project (Debug component) was 85 bug reports and 80 bug reports for the Firefox project.

As explained in Section III, our approach has two phases. In the first phase, the set of files that will be fixed for the bug is predicted. In the second phase, the most appropriate developer for fixing the bug is recommended based on the predicted set of files. The accuracy of the first phase influences the accuracy of the set of recommended developers. Section IV-A provides an evaluation of the accuracy of our location prediction phase, and Section IV-B provides an evaluation of our developer recommendations.

A. Evaluation of Location Prediction Phase

When we predict the files that will be fixed for a bug report using our approach, there are three possible outcomes. First, we correctly predict a source code file that will be changed to fix the new bug report. For our Eclipse and Firefox test sets, we correctly predicted one or more files for 42% of the JDT reports and 62% of the Firefox reports when recommending five locations for a new bug report. If we correctly predict the files to be fixed, then there is a high probability of recommending the correct developer to fix the new bug. Table IV shows the accuracy for predicting the file(s) for our test set of bug reports when recommending up to five files. As expected, we can see that as the number of recommendations increase so does the accuracy for predicting the correct files.

Alternatively, we may incorrectly predict the source code files for the new bug, however there is a meaningful relationship between the files that will be fixed and the predicted file(s) [26]. It is common that when a developer adds a new feature to a project or fixes a bug, she changes multiple files at the same time. This set of changed files (i.e. change set) usually has the same commit message, and sometimes similar words are used in the identifiers. Because of the similar vocabularies in the source code files that are committed together, our approach has a high probability of pointing to the correct portion of the project. This still leads to recommending the correct developer. For example, the actual file changed for bug 100233 is "/org.eclipse.jdt.ui/ui/org/eclipse/jdt/internal/ui/text/correction /JavadocTagsSubProcessor.java". Our approach predicted the file "/org.eclipse.jdt.ui/ui/org/eclipse/jdt/internal/ui/text/correction/JavaCorrectionAssistant.java". Table III shows the nouns extracted from the bug report by our approach and the corresponding weights of these nouns for the two files. Although the predicted file was not the actual file changed for the bug, further examination showed that the most recent commit for the actual file included the predicted file in the change set. This indicates that there is a strong relationship between the actual and predicted file. We also found that the developer that fixed the bug is one of three developers that fixed a recent bug involving the predicted file.

We found that for 49% of the reports in our JDT test set and for 46% of the reports in our Firefox test where the actual source code location was not predicted correctly, at least one of predicted files appeared in a change set of a previous fix containing the correct file.

Lastly, if the set of predicted files contains neither a correct file nor a related file, then it is unlikely that our approach will make a correct developer assignment recommendation.

TABLE III
WEIGHT OF NOUNS IN THE ACTUAL AND PREDICTED FILE FOR ECLIPSE BUG #100233

Used nouns in the Bug	Weight of noun in actual file	Weight of nouns in predicted file
Annotation	-	3
source	-	3
editor	-	2
error	-	2
hierarchy	-	2
marker	-	2
type	3	2
parameter	2	2
problem	-	1
string	2	1
method	3	-
Tag	3	-
code	2	-
List	2	-
Total	**17**	**20**

TABLE IV
AVERAGE ACCURACY PREDICTING CHANGED FILES FOR THE JDT AND FIREFOX PROJECTS.

	JDT	Firefox
Top 1	14.32%	19.51%
Top 2	17.64%	35.37%
Top 3	24.70%	43.90%
Top 4	31.76%	53.66%
Top 5	42.35%	62.20%

TABLE V
AVERAGE OF ACHIEVED FIXER ACCURACIES OF THE APPROACH FOR THE ECLIPSE JDT AND MOZILLA FIREFOX PROJECTS.

	JDT	Firefox
Top 1	48.23%	47.56%
Top 2	61.17%	54.88%
Top 3	81.17%	56.10%
Top 4	88.23%	58.54%
Top 5	89.41%	59.76%

B. Evaluation of Developer Recommendations

As mentioned in Section III-D, the set of recommended developers is the set of developers who have most recently changed one or more of the files predicted by the first phase of our approach. We evaluated the accuracy of our approach for lists of recommendations ranging from one recommendation to five recommendations. Table V shows the results of this evaluation. From the table we can see that the approach has an accuracy around 50% for both projects when recommending one developer. Also, as one would expect, the accuracy improves with more recommendations. This suggests that our assumptions regarding the prediction of the location(s) of bugs to determine the appropriate developers are valid.

Moreover, that the accuracy improves with increasing number of recommendations suggests that we are capturing cases where the best developer to resolve the bug was not chosen and the actual fixer of bug was a sub-optimal choice. Consider for example, a situation where the most suitable developer for fixing a bug abstains from fixing the bug due to a high workload and another developer, also having the necessary expertise, is assigned and resolves the report. Assuming that the recommendations truly reflect the expertise of the developers, then in such a case, the actual fixer of the bug would be ranked second in the recommendations and would be found when recommending two or more developers.

C. Comparison to a Location-Based Approach

We also compared our bug report assignment approach to a location-based approach using information resources that are similar to ones used by our approach, that of Kagdi et al. [27].

Kagdi et al. evaluated their developer recommendations for three levels of fix location: file, package and system. If the predicted file(s) for the bug have not changed in a very long time, or have been recently added, their approach cannot recommend the developer at the file level, and they recommend a developer who is an expert in the package of the predicted file instead (i.e. a package level recommendation). If no package expert can be identified, their approach recommends a system expert. They evaluated their approach using data from three versions of the Eclipse project. As the aim of our approach is to recommend the actual developer who fixed a bug, we compare our results with the file level results of the Kagdi's approach. Table VI shows the accuracy of our approach compared to that of Kagdi for the same test sets. Moreover, this table shows the improvement in accuracy by our approach for each test set.

TABLE VI
COMPARISON OF REACHED ACCURACY BY KAGDI ET AL. [27] AND OUR APPROACH

	Eclipse 2.0	Eclipse 3.0	Eclipse 3.3.2
Kagdi et al. [27]	13.6%	15.7%	27.9%
Our approach	58.14%	85.71%	78.58%
Accuracy improvement	44.54%	70.01%	50.68%

V. THREATS TO VALIDITY

In this section we describe some of the threats to the validity of this work, specifically threats to the internal and external validity.

A. Internal Validity

To populate our file-developer index, we used the "Line-10 Rule" on the most recent source revision commit for a file. However, there may have been cases where this value was not a person with expertise in that area of the source code. For example, the person that fixed the bug may not have commit rights and another project member may be required to commit the fix. However, we did not see any evidence that this was the case in our data set.

This work focused on the use of source code locations for recommending developers. However, bug report assignment is a complex decision involving many factors, such as developer interest (especially in an open source project), developer workload, and the scheduling of developers (e.g. vacations or leave time). It may be the case that the developer who actually fixed a bug according to the the "Line 10 Rule" is not the optimal choice for that report, whereas we treat it as such. Similarly, if source files of the project are changed similtanously by developers, our approach may not accurately captures this behavior as we use only the most recent change.

B. External Validity

The proposed approach in this paper was evaluated using two popular open source projects. The selected projects for evaluating the approaches are representative of the development processes used among large open source software projects. Therefore, we believe that our results will extend to similar projects. However, without further evaluation using other software projects, such as smaller projects or closed source projects, we cannot confirm this belief.

VI. RELATED WORK

Recall that approaches for automatic bug report assignment can be categorized in two ways. Either an approach uses activity information about the project members to make a recommendation (i.e. "activity-based") or the approach uses fault location information (i.e. "location-based").

We first discuss some prior activity-based approaches before discussing related work on location-based approaches. As location-based bug report assignment approaches are similar to those used for bug localization and impact analysis, we also discuss some work from this area.

A. Activity-Based Approaches

The Expertise Browser system by Mockus and Herbsleb [28] used information from source revision commits to determine the expertise of developers for source code files in a software development project. Developers were then ranked for a specific file based on the number of commits they had made that contained the file. We similarly use developer source code activity for making a recommendation, however we restrict our focus to that of using bug fixing activities and not general software development activities.

Unlike approaches that use the summary and description of previously fixed bug reports to make assignment recommendations for a new bug report, Matter et al. [11] determined the expertise of developers based on the vocabulary used in source code. To make an assignment recommendation, the extracted vocabulary was compared with extracted vocabularies from a new bug report. Their approach used information retrieval to weight and determine the relationships between the two extracted vocabularies. This approach is similar to ours in that we also use vocabulary extracted from source code files. However we also use other information sources.

Cubranic and Murphy [7] approached the problem of automatic bug assignment as a text classification problem. They used a Naive Bayes algorithm to create a classifier using a set of previously assigned bug reports, and then used the classifier to recommend an assignment for a new bug report. Anvik et al. [2] extended this approach with further filtering of the data and evaluated the use of various machine learning algorithms for automatic bug report assignment. They found the SVM algorithm to have the best performance for this problem. Unlike these approaches that use a machine learning approach to selecting and weighting terms, our approach uses NLP techniques for extracting terms from information resources, as well as a new term weighting method.

Bhattacharya et al. [12] used a combination of machine learning tools and a probabilistic graph-based model to predict the most appropriate developer for fixing a new bug. They investigated the results of using various machine learning algorithms on data from different projects. They showed that the choice of the best machine learning algorithm is dependent on the quality of the bug reports and varies from project to project. Unlike their approach, our approach selects the most appropriate developer based on their bug fixing activities for source code files.

B. Location-Based Approaches

McDonald and Ackerman [23] designed a tool coined as the "Expertise Recommender" (ER) to locate developers with the desired expertise using vector-based-similarity. The tool uses a heuristic that considers the most recent modification date when developers modified a specific module. Similar to the ER, we recommend the developer that most recently changed one or more of the predicted files.

Canfora and Cerulo [10] presented an approach "aimed at predicting impacted source files and selecting the best candidate developers". They used information found in the comments of source revision commits and the description of the bug report to select the most appropriate developer to assign a new bug report. In their approach, they used the indexes of terms that linked to files of the project and developer names for recommending the fixer. Although both their approach and our approach use similar information resources, they differ in their methods for analyzing the information resources.

Kagdi et al. [29], [27] used an information retrieval-based concept location technique to recommend the most appropriate developers to fix a new change request. They used this technique to establish the relationship between the reported bug and the source code of the project. After the site of the bug in the source code is determined, an assignment recommendation is made based on the predicted location of the bug. This approach is the most similar to our approach. However, our approach does not use IR techniques for predicting the location of the bug and uses a new term weighting method.

The location-based approach presented by Linares-Vasquez et al. [30] used Latent Semantic Indexing (LSI). The developer is recommended based on the authors listed in header comments of the predicted location(s) for the bug. Instead of the complex LSI method, our approach uses simple NLP techniques for predicting bug location and uses developer's bug fixing activity on predicted files to make a recommendation.

C. Bug Localization and Impact Analysis

Rao et al. [13] used Latent Dirichlet Allocation (LDA) for predicting the location of a newly reported bug. Like our approach, they also used source code comments and identifiers as information resources for predicting the locations of bugs, although we also used additional information sources.

Zhou et al. [14] proposed a revised Vector Space Model (VSM) approach for improving the performance for bug localization. Their approach was based on the idea that bugs are more likely to appear in larger files. Like our approach, they improved accuracy by determining the similarity between the text of new bug report and previous fixed bugs.

VII. Future Work

To answer the question "Can the proposed approach correctly recommend the fixer of a bug report?" we used accuracy as our evaluataion metric. However, recommender systems are traditionally evaluated using the measures of precision and recall. Accurately computing these values requires gathering information that provides the set of developers that could have fixed a bug, not just the name of the developer that did fix the bug. We plan to perform such an evaluation in the future.

Also, we have shown that the use of information from four different data sources improves the accuracy of a location-based approach; however we have not investigated the relative importance of these information sources. For example, is the accuracy of the approach more dependant on the terms extracted from bug reports or source code? We plan to conduct an evaluation to determine if such dependencies exist.

Finally, we believe that the use of meta-data in the weighting of the noun terms may improve the accuracy of approaches

such as ours. For example, noun terms that are used in the last two months may be more relevant than terms used a year ago. We plan to investigate this idea in the future.

VIII. Conclusion

In this paper, we presented a location-based approach to automatic bug report assignment that uses four information resources. The use of the different information sources reduces the problems caused by the lack of one information source. We showed that using only noun terms and simple term weighting not only improves the accuracy of a location-based approach, but also avoids the need of some general text analysis steps such as dimensionality reduction and threshold determination. Recommending developers based on source code location further ensures that the selected developers have the necessary experience. Moreover, limiting the recommendations to those developers who have previously fixed bugs at the source code locations was found to improve the overall accuracy of the recommendations. The approach was evaluated using data from the Eclipse and Mozilla projects. When five developers were recommended, our approach had an accuracy of 89.41% and 59.76% for five recommendations on our Eclipse and Mozilla data sets, respectively.

References

[1] 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*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 111–120.

[2] J. Anvik and G. C. Murphy, "Reducing the effort of bug report triage: Recommenders for development-oriented decisions," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 10:1–10:35, Aug. 2011.

[3] K. Crowston, J. Howison, and H. Annabi, "Information systems success in free and open source software development: theory and measures," *Software Process: Improvement and Practice*, vol. 11, no. 2, pp. 123–148, 2006.

[4] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss, "What makes a good bug report?" *Software Engineering, IEEE Transactions on*, vol. 36, no. 5, pp. 618 –643, sept.-oct. 2010.

[5] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *ICSE '07: Proceedings of the 29th international conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 499–510.

[6] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *ICSE '08: Proceedings of the 30th international conference on Software engineering*. New York, NY, USA: ACM, 2008, pp. 461–470.

[7] D. Cubranic and G. C. Murphy, "Automatic bug triage using text categorization." in *SEKE'04*, 2004, pp. 92–97.

[8] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *ICSE '06: Proceedings of the 28th international conference on Software engineering*. New York, NY, USA: ACM, 2006, pp. 361–370.

[9] O. Baysal, M. Godfrey, and R. Cohen, "A bug you like: A framework for automated assignment of bugs," in *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, May 2009, pp. 297 –298.

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

[11] D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning bug reports using a vocabulary-based expertise model of developers," in *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, May 2009, pp. 131 –140.

[12] P. Bhattacharya, I. Neamtiu, and C. R. Shelton, "Automated, highly-accurate, bug assignment using machine learning and tossing graphs," *J. Syst. Softw.*, vol. 85, no. 10, pp. 2275–2292, Oct. 2012.

[13] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972 – 990, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584910000650

[14] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *Software Engineering (ICSE), 2012 34th International Conference on*, june 2012, pp. 14 –24.

[15] A. E. Hassan and R. C. Holt, "The top ten list: Dynamic fault prediction," in *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ser. ICSM '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 263–272.

[16] D. Izquierdo-Cortazar, A. Capiluppi, and J. Gonzalez-Barahona, "Are developers fixing their own bugs?: Tracing bug-fixing and bug-seeding committers," *International Journal of Open Source Software and Processes (IJOSSP)*, vol. 3, no. 2, pp. 23–42, 2011.

[17] A. Tamrawi, T. T. Nguyen, J. Al-Kofahi, and T. N. Nguyen, "Fuzzy set-based automatic bug triaging: Nier track," in *Proc. 33rd Int Software Engineering (ICSE) Conf*, 2011, pp. 884–887.

[18] G. Capobianco, A. D. Lucia, R. Oliveto, A. Panichella, and S. Panichella, "Improving ir-based traceability recovery via noun-based indexing of software artifacts," *Journal of Software: Evolution and Process*, pp. n/a–n/a, 2012.

[19] S. Sarawagi, "Information extraction," *Found. Trends databases*, vol. 1, pp. 261–377, March 2008.

[20] S. L. Abebe and P. Tonella, "Natural language parsing of program element names for concept extraction," in *Proc. IEEE 18th Int Program Comprehension (ICPC) Conf*, 2010, pp. 156–159.

[21] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Improving the tokenisation of identifier names," in *Proc. 25th European Conf. on Object-Oriented Programming*, ser. LNCS, vol. 6813. Springer, 2011, pp. 130–154.

[22] A. Bachmann and A. Bernstein, "Software process data quality and characteristics: a historical view on open and closed source projects," in *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, ser. IWPSE-Evol '09. New York, NY, USA: ACM, 2009, pp. 119–128.

[23] D. W. McDonald and M. S. Ackerman, "Expertise recommender: a flexible recommendation system and architecture," in *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, ser. CSCW '00. New York, NY, USA: ACM, 2000, pp. 231–240.

[24] J. Anvik and G. C. Murphy, "Determining implementation expertise from bug reports," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, ser. MSR '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 2–.

[25] D. Schuler and T. Zimmermann, "Mining usage expertise from version archives," in *Proceedings of the 2008 international working conference on Mining software repositories*, ser. MSR '08. New York, NY, USA: ACM, 2008, pp. 121–124.

[26] R. Shokripour, M. Khansari, and Z. M. Kasirun, "Automatic bug assignment using history of packages," in *ICSIE 2011: Proceedings of the 2011 International Conference on Software and Information Engineering*. Kuala Lumpur, Malaysia: ASME, June 2011, accepted.

[27] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Hammad, "Assigning change requests to software developers," *Journal of Software Maintenance and Evolution: Research and Practice*, 2011.

[28] A. Mockus and J. D. Herbsleb, "Expertise browser: a quantitative approach to identifying expertise," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 503–512.

[29] H. Kagdi and D. Poshyvanyk, "Who can help me with this change request?" in *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, may 2009, pp. 273 –277.

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