Triaging Incoming Change Requests: Bug or Commit History, or Code Authorship?

Mario Linares-Vásquez[1], Kamal Hossen[2], Hoang Dang[2], Huzefa Kagdi[2], Malcom Gethers[1], Denys Poshyvanyk[1]

[1]Computer Science Department
The College of William and Mary
Williamsburg, VA 23185
{mlinarev, mgethers, denys}@cs.wm.edu

[2]Department of Computer Science
Wichita State University
Wichita, KS 67260-0083
{mxhossen, hndang, huzefa.kagdi }@wichita.edu

***Abstract* — There is a tremendous wealth of code authorship information available in source code. Motivated with the presence of this information, in a number of open source projects, an approach to recommend expert developers to assist with a software change request (e.g., a bug fixes or feature) is presented. It employs a combination of an information retrieval technique and processing of the source code authorship information. The relevant source code files to the textual description of a change request are first located. The authors listed in the header comments in these files are then analyzed to arrive at a ranked list of the most suitable developers. The approach fundamentally differs from its previously reported counterparts, as it does not require software repository mining. Neither does it require training from past bugs/issues, which is often done with sophisticated techniques such as machine learning, nor mining of source code repositories, i.e., commits.**

An empirical study to evaluate the effectiveness of the approach on three open source systems, *ArgoUML*, *JEdit*, and *MuCommander*, is reported. Our approach is compared with two representative approaches: 1) using machine learning on past bug reports, and 2) based on commit logs. The presented approach is found to provide recommendation accuracies that are equivalent or better than the two compared approaches. These findings are encouraging, as it opens up a promising and orthogonal possibility of recommending developers without the need of any historical change information.

Keywords - code authorship; information retrieval; change request; triaging; expert developer recommendations

I. Introduction

Software change requests, such as bug fixes and new features, are an integral part of software evolution and maintenance. Effectively supporting software changes is essential to provide a sustainable high-quality evolution of large-scale software systems. One of the change management issues that has gained a wide attention in the last few years is the automatic support for recommending expert developers to address change requests [1, 3, 18-20, 24-26, 28, 29, 34, 37]. Change requests are typically specified in a free-form textual description using natural language (*e.g.*, a bug reported to the *Bugzilla* system of a software project). It is not uncommon in such projects to receive tens of change requests daily that need to be resolved in an effective manner (*e.g.*, within time, priority, and quality factors). Therefore, assigning change requests to the developers with the right implementation expertise is challenging, but certainly a much needed activity.

A number of approaches have been proposed to help identify developers with the software maintenance task at hand [1, 3, 18-20, 24-26, 28, 29, 34, 37]. At the change request, there are broadly two types of approaches to recommend developers to handle incoming change requests: 1) building a model that trains from the past bug reports using their descriptions and developers who were assigned to them [1, 2], and 2) using a combination of a concept location technique to locate relevant source code to a bug request and then mine the source code (commit) repository to recommend developers [19, 21]. Both these approaches require extensive mining of software repositories.

We present a novel approach to developer recommendation that does not require mining of either a bug or commit repository. Central to our approach is the use of the author information present in the source code files. Authors are typically found in the header comments of the source code entities (*e.g.*, file, class, method). Figure 1 shows the author *mvw* is found in first line of the header comment of the file *OperationNotationUml.java.* Authors *mvw@tigris.org* and *jaap.branderhorst@xs4all.nl* are found in the header comments of the class *OperationNotation Uml.java* and its method *toString()*, see the highlighted red boxes. The premise of our technique is that the authors of source code entities are best equipped to tackle any changes needed in them. This authorship information can be leveraged once relevant source code, to a change request, is located. Therefore, we first employ an Information Retrieval (IR) based concept location technique [12] to find relevant code entities to a given change request. The authorship information in these source code entities is then used to recommend a ranked list of developers.

To evaluate the accuracy of our technique, we conducted an empirical study on three open source systems: *ArgoUML*, *jEdit*, and *MuCommander*. Precision and recall values of the developer recommendations on a number of bug reports sampled from these systems are presented. That is, how effective our approach is at recommending the actual developer who ended up fixing these bugs. Additionally, our authorship-based approach is empirically compared with two other approaches that require mining of software repositories. The results show that our new approach

978-1-4673-2312-3/12/$31.00 ©2012 IEEE

performs as well as, or better than, the two other competitive approaches in terms of recommendation accuracy.

Our paper makes the following noteworthy contributions:

- A novel developer recommendation approach for incoming change request that is centered on the code authorship information. Our approach is lightweight, as it does not require software repository mining. To the best of our knowledge, there is no other such approach in the literature.
- A comparative study of our approach with two other approaches that are based on mining of software repositories. The results show that our lightweight approach can perform equally well, or better than, the heavy weight mining approaches.

II. CODE AUTHORSHIP BASED APPROACH TO DEVELOPER RECOMMENDATION

Our approach to triaging incoming change requests consists of the following two steps:

1. Given a change request description, we use Latent Semantic Indexing (LSI) [10] to locate a ranked list of relevant units of source code (*e.g.*, files, classes, and methods) that match the given description in a version of the software system. This version is typically the one in which an issue is reported or a snapshot of source code before the change request is implemented (e.g., bug is fixed).
2. The authors of the units of source code from the above step are then analyzed to recommend a ranked list of developers to deal with those units (*e.g.*, classes). Here, authors are the developers listed in the source code files, typically in the header comments of entities (files, classes, and/or methods).

A. *Locating Relevant Files with Information Retrieval*

In our approach, in order to locate textually relevant files, we rely on an IR-based concept location technique [31]. This technique can be summarized in the following steps:

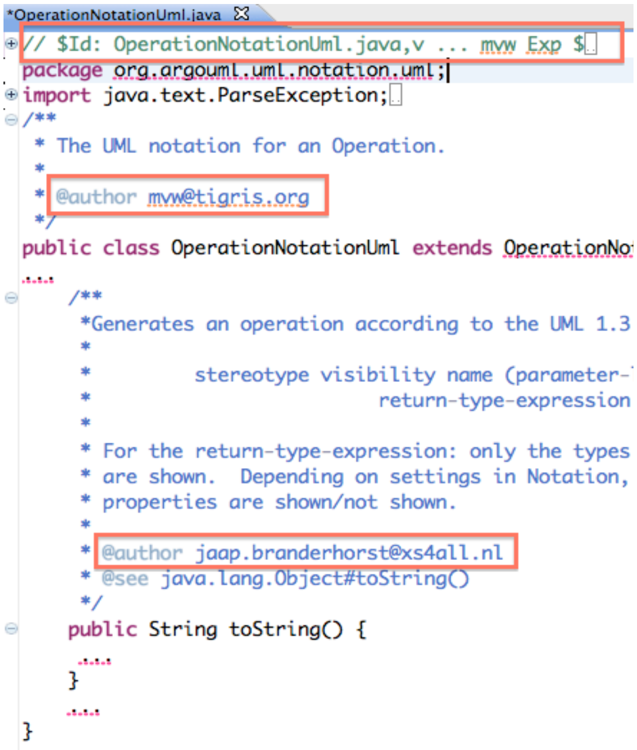
1. **Creating a corpus from software**: The source code is parsed using a developer-defined granularity level (*i.e.*, *files*) and documents are extracted from the source code. A corpus is created so that each file will have a corresponding document in the resulting corpus. Only identifiers and comments are extracted from the source code.
2. **Indexing a corpus:** The corpus is indexed using LSI and its real-valued vector subspace representation is created. Dimensionality reduction is performed in this step, capturing the important semantic information about identifiers and comments and their latent relationships. In the resulting subspace, each document has a corresponding vector. The steps 1 and 2 are performed offline once, while 3 and 4 are repeated for a number of open change requests.
3. **Using change requests:** A set of words that describes the concept of interest constitutes the initial query. We used long descriptions of change requests, *i.e.*, the long description of a bug or a feature given by the developer or reporter in the bug tracking system. We did not use the follow-up comments. This query is used as an input in the step 4 to rank the documents.
4. **Rank documents:** Similarities between the user query (i.e., change request) and documents in the corpus are computed. The similarity between a query reflecting a concept and a set of data about the source code indexed via LSI allows for the generation of a ranked list of documents relevant to that concept. All the documents are ranked by the similarity measure in descending order (*i.e.*, the most relevant at the top and the least relevant at the bottom).

B. *Using Authorship to Recommend Expert Developers*

The basic premise of this approach is that the developers who are listed as authors in the source code files are likely to best assist with their current or future changes. It is not uncommon to have such authorship information available in the open source development paradigms. Figure 1 shows an example of a source code file and the author information therein from the *ArgoUML* project.

The specifics of our approach are as the following:

1. **Obtaining source code files**: The source code of the top relevant files that are retrieved by the concept location component of our technique is first obtained. These files are derived from the same release used by the concept location component of our technique.
2. **Converting files to *srcML* representation**: The source code files in the above step are converted to the *srcML*-based representation. *srcML* is a lightweight XML representation for *C/C++/Java* source code with selective *Abstract Syntax Tree* information embedded [9]. This conversion is done for the ease of extraction of comments from the source code. We use *srcML* here; however, this element of our approach can be easily replaced by any lightweight source code analysis method, including regular expressions.
3. **Extracting header comments:** All the header comments are extracted from all the *srcML* files via a straightforward XML processing. The header comments are generally the first comment in a source code file, source code classes, and/or methods. The header comments typically contain the copyright, licensing, and authorship information. Additionally, it may also contain information about the (last version) change, automatically inserted with a keyword expansion mechanism from version-control systems. Figure 1 shows that the author *mvw* is found in first line of the header comment of the file *OperationNotationUml.java.* Authors *mvw@tigris.org* and *jaap.branderhorst@xs4all.nl* are found in the header comments of the class *OperationNotationUml.java* and one of its methods, see the highlighted areas in red.
4. **Extracting authors from comments:** The content and format of the author listing in the header comments may vary across systems. From a thorough manual examination of a number of open source projects, we devised regular expressions to extract the authors from the header comments. Authors are extracted from each



```
// $Id: OperationNotationUml.java,v ... mvw Exp $
package org.argouml.uml.notation.uml;
import java.text.ParseException;
/**
 * The UML notation for an Operation.
 *
 * @author mvw@tigris.org
 */
public class OperationNotationUml extends OperationNo
....
    /**
     *Generates an operation according to the UML 1.3
     *
     *          stereotype visibility name (parameter-
     *                         return-type-expression
     *
     * For the return-type-expression: only the types
     * are shown.  Depending on settings in Notation,
     * properties are shown/not shown.
     *
     * @author jaap.branderhorst@xs4all.nl
     * @see java.lang.Object#toString()
     */
    public String toString() {
     ....
    }
    ....
}
```

Figure 1. A snipped of the file *OperationNotationUml.java* from*ArogUML*. The author *mvw* is found in the header comment of the file, and the authors *mvw@tigris.org* and *jaap.branderhorst@xs4all.nl* are found in the header comment of the class *OperationNotationUml* and method *toString*(), which are all highlighted in red boxes.

of the relevant files produced by the concept location component. Note that the same developer could have multiple identities. We extracted and compiled all the entities of each developer from the project resources, and mapped them to a unique identifier. To match author to change requests, we collected names, email addresses, and user IDs from the project's *Bugzilla*, *Subversion*, and web site. A large percentage of identities were matched automatically via string matching or with heuristics (e.g., IDs were a part of email-ids or abbreviated from names). About 15-20% of identities required a manual mapping. For example, the identities *Michiel van der Wulp* (full name), *mvw@tigris.org* (email address) and *mvw* (user name) represent the same developer, and this developer is mapped to the identity *mvw*. Similarly, the identities *jaap.branderhorst@xs4all.nl* and *jaap* represent the same developer, and this developer is mapped to the identity *jaap*.

5. **Ranking authors:** There is a one-to-many relationship between an IR query, i.e., description of a bug b_i, and source code files. Given a user provided cutoff point of n, we get the n top ranked source code files $f_1, f_2, \ldots, f_n$ for the bug b_i. Also, there is a one-to-many relationship between the source code file and authors. That is, each file f_i may have multiple authors; however, it is not necessary for all the files to have the same number of authors. For example, the file f_1 could have two authors and the file f_2 could three authors. Two files may have common authors.

Given a user provided cutoff point of m, we need to get the top m ranked authors (developers). We use a frequency-based approach to rank authors. The hypothesis is that the higher the occurrence of an author in the relevant files to a change request, the more knowledgeable that author is in handling that particular request. We take a union of all the authors appearing in all the n files. This union gives us a set of cardinality d unique authors. For each author d_i, we count the number of files in which he/she appears. Once a frequency count of each author is obtained, all the authors are sorted in descending order of their file frequency counts. From this sorted list of authors, we recommend the top m ranked authors that are the most likely developers to assist with fixing the bug/change request in question. We break ties using the information of their file ranks and lexical positions in the source code file. That is, if two developers d_1 and d_2 have the same frequency count, we rank the developer d_1 higher if it first appears in a file ranked above the first file in which the developer d_2 is found. If we cannot break the ties with file ranks, we use the developers' first lexical positions in the source code file. That is, the developer d_1 will be ranked ahead of the developer d_2, if both appear in the same file; however, d_1 first appears ahead of d_2 in the source code text. The lexical positions in a way correspond to the file→class→method hierarchy. At the same level, e.g., file, they are picked in the order they appear in the text. The entire authorship-processing step on *ArgoUML*, *jEdit*, and *MuCommander* datasets (see Table) took approximately a minute and a half on a commodity desktop running *Ubuntu*.

C. *An Example from ArgoUML*

Here, we demonstrate the workings of the approach using an example from *ArgoUML*. The change request of interest here is the bug# *4078*. The reporter described it as follows:

> *"Operation box in CallAction proppanel is too small".*

We consider the above textual description to be a concept of interest. We collected the source code of *ArgoUML 0.22* (the bug was not fixed as of this date). We parsed the source code of *ArgoUML* using the class-level granularity (*i.e.*, each document is a class). After indexing with LSI, we obtained a corpus consisting of 1,449 documents and containing 5,488 unique words. We formulated a search query using the bug's textual description, which was used as an input to LSI-based concept location tool. The results of the search (*i.e.*, a ranked list of relevant files) are summarized in Table I.

The contents of the ten files in Table I were processed with our authorship analysis component. Table II shows the authors extracted from each of the ten files listed in Table I. For example, the file *OperationNotationUml.java* is collectively authored by developers mvw and jaap, and the file OperationNotation.java is authored by mvw. The developers in the same file are listed in the lexical order in which they appear in the source code file. For example, the developer mvw first appeared ahead of the developer jaap in the file OperationNotationUml.java.

The results obtained in Table II are input to our frequency-based ranking mechanism to arrive at a ranked list of developers to handle the b*ug# 4078*. Table III shows the ranked list of developers produced after the application of the ranking mechanism. The developer *mvw* ends up at the top because it appears in five files (see Table II for the specific files). The developers *linus* and *euluis* are tied because both appear in one file. The first occurrence of both of these developers in terms of the file location is also tied. Both of them occur first in the file *ModelerImpl.java.* The developer *linus* is given a higher rank than the developer *euluis* because of their lexical orders in the file *ModelerImpl.java*.

The bug# *4078* was fixed by the developer *mvw,* verified with revision# *10060* in the subversion system of *ArgoUML*. As it can be clearly seen, this developer was ranked first by our approach (see Table III). The same bug report, when operated with two other mining-software-repositories approaches by Anvik et al. [1] and Kagdi et al. [19], did not yield this correct developer in the top results.

III. Case Study

The purpose of this empirical study was to investigate how well our authorship-based approach recommends expert developers to assist with incoming change requests. We also compared our authorship-based approach (denoted here as *Authorship*) with two previously published approaches. The first approach is based on the mining of a bug report history by Anvik et al. [1]), which we implmented (denoted here as machine learning - *ML*). The second is based on mining of source control repositories, *i.e.*, commit logs, by Kagdi et al. [19] (denoted here as *xFinder*). Therefore, we addressed the following research questions (RQ) in our case study:

> **RQ$_1$:** How does the accuracy of the *Authorship* approach compare to its two competitors that are based on software repository mining, namely *ML* and *xFinder*?
>
> **RQ$_2$:** Is there any impact on the results of *Authorship* when filtering of IR-based results with dynamic-analysis information is included, *i.e.*, an additional analysis cost is incurred?

The rationale behind **RQ$_1$** is two-fold: 1) To assess whether our *Authorship* can identify correct developers to handle change requests in open source systems, and 2) how well the accuracy of the authorship approach compares to the *ML* and *xFinder* approaches.

We used LSI, an IR technique, to locate relevant files to a given change request. Previous studies have shown that such a technique is prone to false positives; for example, it may recommend a file to be relevant when it is not [33]. The purpose of **RQ$_2$** is to assess if incorporating an additional software analysis technique to the first step of the *Authorship* approach improves its accuracy results.

TABLE I. Top ten files relevant to the bug # *4078* in *ArgoUML*

Rank	Files
1	*mdr/CommonBehaviorHelperMDRImpl.java*
2	*uml/OperationNotationUml.java*
3	*common_behavior/PropPanelCallAction.java*
4	*notation/OperationNotation.java*
5	*reveng/ModelerImpl*
6	*ui/FigClassifierBox.java*
7	*java/OperationNotationJava.java*
8	*mdr/MetaTypesMDRImpl.java*
9	*ui/UMLClassDiagram.java*
10	*mdr/CommonBehaviorFactoryMDRImpl.java*

TABLE II. The authors extracted from each of the top ten files relevant to the Bug# *4078*. This is an intermediate result produced by Step 4 of our recommendation approach.

Files	Authors
mdr/CommonBehaviorHelperMDRImpl.java	*tfmorris, rastaman*
uml/OperationNotationUml.java	*mvw,jaap*
common_behavior/PropPanelCallAction.java	*mvw*
notation/OperationNotation.java	*mvw*
reveng/ModelerImpl	*linus,euluis*
ui/FigClassifierBox.java	*tfmorris*
java/OperationNotationJava.java	*mvw*
mdr/MetaTypesMDRImpl.java	*mvw*
ui/UMLClassDiagram.java	*bobtarling, jrobbins*
mdr/CommonBehaviorFactoryMDRImpl.java	*tfmorris,rastaman ,thierrylach*

TABLE III. The final ranked list of developers recommended

Developer ID	File Freq	Developer ID	File Freq
mvw	*5*	*linus*	*1*
tfmorris	*3*	*euluis*	*1*
rastaman	*2*	*bobtarling*	*1*
jaap	*1*	*thierrylach*	*1*

We used a dynamic analysis technique beacause it was found to improve the accuracy of IR-based feature location and impact analysis approaches [16, 30]. That is, we want to study if using the dynamic filtering of IR results within our approach outperforms the accuracy of the *ML*, *xFinder*, and *Authorship* without the dynamic filtering.

Next, we provide background information on the two competitive approaches used in our evaluation.

A. ML on Past Bug Reports for Assigning Developers

To recommend developers, Anvik et al. [1] used a history of previous bug reports from *Eclipse*, *Firefox*, and *gcc* that had been resolved or assigned between September 1, 2004 and May 31, 2005 – training instances. The list of developers assigned to, or resolved, each report was considered the *label* (output field) for the textual documents (input fields). The one-line summary and the full text description of each bug report were considered a document, and their words were considered the attributes that represent the documents. Stops-words and non-alphabetic tokens were removed and the vector representation was built computing the *tf-idf* measure on the remaining words. Neither stemming nor attributes selection methods were applied [1].

In order to compare our authorship approach to this previously published technique, we reproduced the ML-based approach of Anvik et al. [1]. We used the same preprocessing steps (stops-words removal, no stemming, *tf-idf* as a term weighting method, and no attribute selection method). We did not find precise details on the parameters and settings of the algorithms in [1], therefore, we only ran experiments with two implementations of SVM provided by Weka[1] (*SMO* and *LibSVM*) using a linear kernel. We decided to use SVM because it was found to be a superior classifier in several domains, such as text categorization [22], software categorization [27, 39], and developers recommendation [1, 3].

Recommending more than one developer requires *ML* classifiers that provide more than one label for a testing instance. It means that they should be able to deal with multi-label classification problems. Anvik et al. [1] provide results from recommendations with one, two, and three developers. We used the ranking of the SVM classifiers on the labels to build the developer recommendations from top one to ten developers. Therefore, we ran the SVM implementations using a one-against-all strategy to deal with multiple developer recommendations. In this strategy, a classifier is built for each of the developers in the dataset. For example, for a dataset with ten developers, there should be ten SVMs, each SVM is trained to recommend only one developer, and the overall recommendation is built using the recommendations of the ten classifiers. Overall, the ranking of developers is based on the ranking provided by each SVM. Thus, for a top-k recommendation we made the list with the *k* developers with the top-k rankings.

B. *xFinder Approach for Recommending Developers*

xFinder approach to recommending experts to assist with a given change request consists of the following two steps:

1. The first step is identical to the first step of the presented authorship approach (see Section II.B).
2. The version histories of units of source code from the above step are then analyzed to recommend a ranked list of developers that are the most experienced and/or have substantial contributions in dealing with those units (*e.g.*, classes/files).

We used the *xFinder* approach to recommend expert developers by mining version archives of a software system [20]. The basic premise of this approach is that the developers who contributed substantial changes to a specific part of source code in the past are likely to best assist in its current or future changes. This approach uses the commits in repositories that record source code changes submitted by developers to the version-control systems (*e.g.*, *Subversion*[2] *and CVS*). *xFinder* considers the following factors in deciding the expertise of the developer *d* for the file *f*:

- The number of commits, *i.e.*, commit contributions that include the file *f* and are committed by the developer *d*.
- The number of workdays, *i.e.*, calendar days, of the developer *d* with commits that include the file *f*.
- Most recent workday in the activity of the developer *d* with a commit that includes the file *f*.

We used the source code commits of the three systems from the history period before the releases that were chosen for the LSI indexing to train *xFinder*.

TABLE IV. SUBJECT SOFTWARE SYSTEMS USED IN THE CASE STUDY

System	Ver.	LOC	Files	Methods	Terms
jEdit	4.3	103,896	503	6,413	4,372
ArgoUML	0.22	148,892	1,439	11,000	5,488
muCommander	0.8.5	76,649	1,069	8,187	4,262

TABLE V. SUMMARY OF THE BENCHMARKS

System	# Change requests	Developers in gold set: descriptive stats		
		Min	*Mean*	*Max*
jEdit	143	1	1.06	2
ArgoUML	91	1	1.05	2
muCommander	92	1	1.01	2

C. *Subject Software Systems*

The *context* of our study is characterized by three open source *Java* systems, namely *jEdit v4.3*, a popular text editor, *ArgoUML v0.22*, a well-known UML editor, and *muCommander v0.8.5*, a cross-platform file manager. The sizes of these considered systems range from 75K to 150K LOC and contain between 4K and 11K methods. The stats of these systems are detailed inTable IV.

D. *Building the benchmarks*

For each of the subject systems, we created a benchmark to evaluate our *Authorship* approach and compare it with *ML* and *xFinder*. The benchmark consists of a set of change requests that has the following information for each request: a natural language query (request summary) and a gold set of developers that addressed each change request.

The benchmark was established by a manual inspection of the change requests (done by one of the authors), source code, and their historical changes recorded in version-control repositories. *Subversion* (SVN) repository commit logs were used to aid this process. For example, keywords such as *Bug Id* in the commit messages/logs were used as starting points to examine if the commits were in fact associated with the change request in the issue tracking system that was indicated with these keywords. The author and commit messages in those commits, which can be readily obtained from SVN, were processed to identify the developers that contributed changes to the change requests, *i.e.*, goldset, which forms our actual developer set for evaluation. The details on the change requests are summarized in Table V. Also, the minimum, mean, maximum number of developers for the consdiered change requests are presented. As it can be seen, a vast majority of change requests are handled by a single developer (i.e., commit contributors). In some cases, we found the commiter was different from the acutal developer who contributed changes. The actual developer was mentioned in the commit comments, we included that developer in our goldset.

[1] http://www.cs.waikato.ac.nz/ml/weka/ (verified on 04/22/12)

[2] http://subversion.tigris.org/ (verified on 04/22/12)

Our technique operates at the change request level, so we also need input queries to test. These queries were constructed by concatenating the title and the description of the change requests referenced from the SVN logs.

E. Collecting and Using Execution Information

The idea of integrating IR with dynamic analysis was previously defined in the context of feature location [23]; however, it was not used to improve the bug triaging before. A single feature or bug-specific execution trace is first collected. *IR* then ranks all the methods in the trace instead of all the methods in a software release. Therefore, the run-time information is a filter that eliminates files based on methods that were not executed and are less likely to be relevant to the change request. The dynamic information, if and when available, can be used to eliminate some of the false positives produced by *IR* [16, 30]. We denote a version of our approach that uses execution information as *Authorship*$_F$. Similarly, the version of *xFinder* that uses execution information is denoted as *xFinder*$_F$. We also included the dynamic filtering in *xFinder* to enable a fair comparison. Further details on how we collected execution information can be found elsewhere [16]..

F. Metrics and Statistical Analyses

We evaluated the accuracy of each one of the approaches, for all the reports in our testing set, using the same precision and recall metrics of Anvik et al. [1]. The formulae for these metrics are listed below:

$Precision = |Rec\_devs \cap Actual\_devs| \;/\; |Rec\_devs|$

$Recall = |Rec\_devs \cap Actual\_devs| \;/\; |Actual\_devs|$

These metrics were computed for recommendation lists of developers with different sizes (ranging from the top one developer to ten developers). To analyze the differences between the values reported by each approach, we computed the average values on each dataset and compared them using a precision-recall chart. Moreover, we applied the *Mann-Whitney* test to validate whether there was a statistically significant difference with α=0.05 between the results. We used this non-parametric test because we did not assume normality in the distributions of precision and recall results. This test assesses whether all the observations in two samples are independent of each other [17]. The other purpose of the test is to assess whether the distribution of one of the two samples is stochastically greater than the other. Therefore, we defined the following null hypotheses for our study (we do not list alternative hypotheses, but they should be easy to derive from these null hypotheses respectively):

$\mathbf{H_{0-1}}$: There is no statistically significant difference between the **precision/recall** of *ML* and *Authorship*.

$\mathbf{H_{0-2}}$: There is no statistically significant difference between the **precision/recall** of *xFinder* and *Authorship*.

$\mathbf{H_{0-3}}$: There is no statistically significant difference between the **precision/recall** of *xFinder*$_F$ and *Authorship*.

$\mathbf{H_{0-4}}$: There is no statistically significant difference between the **precision/recall** of *ML* and *Authorship*$_F$.

$\mathbf{H_{0-5}}$: There is no statistically significant difference between the **precision/recall** of *xFinder* and *Authorship*$_F$.

$\mathbf{H_{0-6}}$: There is no statistically significant difference between the **precision/recall** of *xFinder*$_F$ and *Authorship*$_F$.

$\mathbf{H_{0-7}}$: There is no statistically significant difference between the **precision/recall** of *Authorship*$_F$ and *Authorship.*

The hypotheses from $\mathbf{H_{0-1}}$ up to $\mathbf{H_{0-6}}$ were used to answer $\mathbf{RQ_1}$, and $\mathbf{H_{0-7}}$ was used to answer $\mathbf{RQ_2}$.

G. Case Study Results

Figure 2 depicts the average precision and recalls for the three systems[3]. For top-1 recommendations, we found that *Authorship* provided the highest values of precision and recall for *ArgoUML*, and *xFinder*$_F$ provided the highest values of precision and recall for *JEdit* and *MuCommander*. However, the behavior for recommendations with more developers is different. For example, *ML* had the best accuracy from top-2 to top-10 in the *ArgoUML* dataset; *xFinder and xFinder*$_F$ had the best accuracy from top-1 to top-10 developers in *JEdit*; *Authorship* had the best accuracy for *MuCommander* from top-2 to top-10.

For top-1 recommendations in *ArgoUML* (Figure 2.a), the *Authorship* provided the highest accuracy. However, we did not found statistical significant difference between the accuracies of the *Authorship* and the other techniques. One possible explanation is that the acceptable precision values for top-1 recommendations are either zero or one, and the *Authorship* had a precision of one in 46 times, while *ML* had a precision of one in 29 cases. Although the difference between the precision of the *Authorship* and *ML* is 19%, the distribution of zeros and ones in both approaches is very similar.

For *ArgoUML*, from top-2 to top-10, the other approaches outperformed the precision and recall reported by the *Authorship* technique with a significant difference from top-3 to top-10 (except for top-3 recall, top-8 recall, and top-10 precision). The difference in precision from top-2 to top-10 for *xFinder* vs. *Authorship* ranged from 0.8% to 4% with mean 2.5%, and for *ML* vs. *Authorship* ranged from 0.3% to 6% with mean 3.4%; the difference in recall from top-2 to top-10 for *xFinder* vs. *Authorship* ranged from 4.9% to 21.4% with mean 15.8%, and for *ML* vs. *Authorship* ranged from 4.4% to 24.7% with mean 19.1%. The reason behind this sharp decline in the *Authorship* performance is due to the fact that the top-1 precision is almost twice compared to the other techniques. Also, only a single developer handles each of the change requests in the benchmark. Increasing the recommendations from top-1 to top-2 added the second recommendation as a false positive. Therefore, adding an extra recommendation did not help improve the precision.

[3] We only report the results of SMO with parameter C = 1 and non-standardized data because this SVM implementation in Weka is akin to the SMV used by Anvik et al.

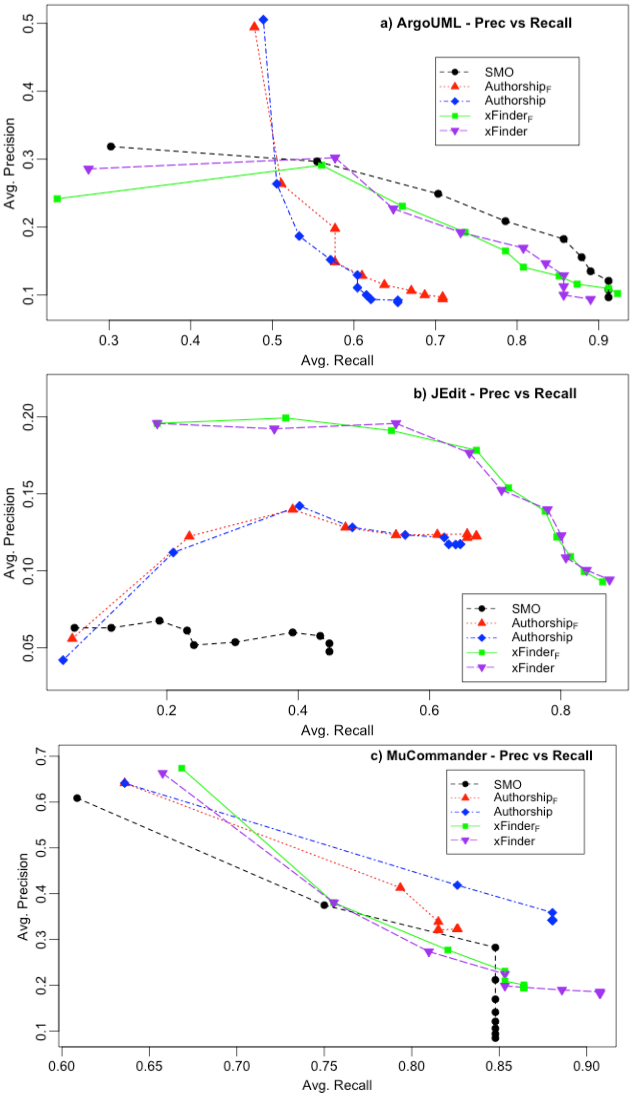


Figure 2. Precision vs. recall charts for *ArgoUML*, *JEdit*, *MuCommander*. These results are for four approaches (*ML* – SMO, *xFinder*, *xFinder*$_F$, *Authorship,* and *Authorship*$_F$). Each curve has a point for each recommendation from top-1 to10.

For *JEdit* (Figure 2.b), *xFinder* had a higher accuracy than *ML* and *Authorship* from top-1 to top-10 recommendations with a significant difference (except for top-5 and top-6 precision). *Authorship* exhibited higher accuracy than *ML* from top-2 to top-10 recommendations with a difference from top-3 to top-10; the difference in precision from top-2 to top-10 for *Authorship* vs. *ML* ranged from 5.9% to 7.5% with mean 6.8%, and the difference in recall ranged from 11.9% to 30.8% with mean 23.5%.

For *MuCommander* (Figure 2.c), the *Authorship* showed higher precision values than *ML* and *xFinder* from top-2 up to top-10 recommendations with a statistical significant difference (except for top-2). We found that the difference in precision from top-2 to top-10 for *Authorship* vs. *ML* ranged from 3.8% to 23.8% with mean 15.8%.

The *Authorship* outperformed precision and recall of *ML* in *JEdit* and *MuCommander*. We found significant differences between the precisions of the two approaches in recommendations from the top-3 to top-10 developers, on *JEdit* and *MuCommande*r (Table VI). Therefore, for **RQ$_1$** we concluded that **the precision of the *Authorship* outperformed *ML* on *JEdit* and *MuCommander* datasets.**

TABLE VI. Heat-map summarizing results for testing hypotheses across all the systems. The color in each cell represents the number of times the Man-Whitney test suggested statistically significant difference: black cells mean that the test found significant difference across all the three datasets; dark-gray – two out of three systems; light-gray – one system; white – no significant difference in all the three systems.

H		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
H_{0-1}	P	□	░	■	■	■	■	■	▓	■	■
H_{0-1}	R	□	░	░	▓	▓	▓	▓	▓	▓	▓
H_{0-2}	P	▓	░	▓	▓	░	░	░	▓	▓	▓
H_{0-2}	R	▓	░	░	░	▓	▓	▓	▓	▓	▓
H_{0-3}	P	▓	░	▓	▓	▓	░	░	▓	▓	■
H_{0-3}	R	▓	░	░	░	▓	▓	▓	▓	░	▓
H_{0-4}	P	▓	□	■	■	■	■	■	▓	■	▓
H_{0-4}	R	▓	□	░	▓	▓	▓	▓	▓	▓	▓
H_{0-5}	P	▓	░	▓	▓	░	░	▓	▓	▓	▓
H_{0-5}	R	▓	░	░	▓	▓	▓	▓	▓	▓	▓
H_{0-6}	P	▓	░	▓	▓	■	░	▓	▓	▓	▓
H_{0-6}	R	▓	░	░	░	▓	▓	▓	▓	▓	▓
H_{0-7}	P	□	□	□	□	□	□	□	□	□	□
H_{0-7}	R	□	□	□	□	□	□	□	□	□	□

Authorship also outperformed precision of *xFinder* in *MuCommander*. We found significant differences between the precisions of the two approaches in recommendations from the top-3 to top-10 developers. Therefore, for **RQ$_1$**, we concluded that **the precision of the *Authorship* outperformed *xFinder* on *MuCommander*.**

For **RQ$_2$,** we did not find a conclusive support for a significant difference in the accuracies of *Authorship and Authorship*$_F$. We could not reject **H_{0-7}** in any of the systems. Therefore, we concluded that ***Authorship performs as well as Authorship*$_F$ in terms of accuracy**. *These results suggest that the additional overhead of dynamic analysis in the Authorship and xFinder was not justified, as there was no statistically significant accuracy gain.*

Now, we provide representative bugs from the three systems detailing where *Authorship* outperformed the other approaches. For example, *Authorship* achieved a precision of 100% for the bug report# 2129419 in *JEdit* using the first recommendation (top-1), while the highest precision for *xFinder* (50%) was achieved with top-7, and the *ML* could not predict the correct developer (*kpuer*) with any of the recommendations. Other example where *Authorship* provided a better accuracy without recommending a large number of developers, compared to the other approaches, is the bug report# 4031 in *ArgoUML* fixed by the developer "*mvw*" (Michiel van der Wulp). For that report, *Authorship* achieved a 100% precision in the very first recommendation (top-1), while *xFinder* got a precision of 50% with five recommendations, and *ML* was able to achieve only 33% precision within top-3. *Authorship* and *ML* obtained 100%

precision with top-1 recommendation for the bug report# 277 in *MuCommander*, however, the *xFinder$_F$* achieved its highest value of precision of 50% using a recommendation with five developers.

IV. Threats to validity

We identify threats to validity that could influence the results of our empirical study and our conclusions.

A. Construct Validity

We discuss threats to construct validity that concern the means that are used in our method and its accuracy assessment as a depiction of reality. In other words, do the accuracy measures and their operational computation represent correctness of developer recommendations?

Concept location may not find source code exactly relevant to a bug or a feature: The IR-based concept location tool did not exactly return the classes (files) that were found in the commits related to the bug fixes or feature implementations in all the cases. However, it is interesting to note from the accuracy results that the classes that were recommended were either relevant (but not involved in the change that resolved the issue) or conceptually related (*i.e.*, developers were also knowledgeable in these parts).

Accuracy measures may not precisely measure the correctness of developer recommendations: A valid concern could be a single measure of accuracy that was used in our method does not provide a comprehensive picture, *i.e.*, an incomplete and monolithic view of accuracy from the considered dataset. We used two widely used metrics precision and recall in our study. We considered a gold-set of developers who contributed source code changes to address change requests (i.e., fixes). It is possible that there are other developers who are equally capable of resolving these change requests; however, such a gold-set is difficult to ascertain (without involving the project stakeholders, for example). Nonetheless, our undertaken benchmark provides a careful accuracy values (perhaps conservative bounds).

B. Internal Validity

We discuss threats to internal validity that concern factors that could have influenced our results.

Factors other than expertise are responsible for the developers ending up resolving the change requests: In our case study, we showed that there is a positive relationship between the developers recommended with our approach to work on change requests and the developers who fixed them in the software repositories (i.e., considered baseline). It is possible that other factors, such as schedule, work habits, technology fade or expertise, and project policy/roles are equally effective or better. A definitive answer in this regard would require another set of studies.

C. External Validity

We discuss threats to external validity that concern factors that are associated with generalizing the validity of our results to datasets other than considered in our study.

Assessed systems are not representative: The accuracy was assessed on three open source systems, which we believe are good representatives of large-scale, collaboratively developed software systems. However, we cannot claim that the results presented here would equally hold on other systems (*e.g.*, closed source). If the authorship information is not present in the source code files, our approach may not applicable.

Sampled sets of change requests are not sufficient: The size of the evaluation sample and the number of systems remains a difficult issue, as there is no accepted "gold standard" for developer recommendation problem. The approach of "more, the better" may not necessarily yield a rigorous evaluation, as there are known issues of bug duplication [35, 40] and other noisy information in bug/issue databases [4, 5]. Not accounting for such issues may lead to biased results positively or negatively or both. The considered sample sizes in our evaluation, however, is not uncommon, for example, Anvik et al. [1] also considered 22 bug reports from *Firefox* in their evaluation. Nonetheless, this topic remains an important part of our future work.

Accuracy offered by our method may not be practical: We compared the accuracy results of our approach with two other approaches. Our approach is competitive with these approaches or better. We plan to pursue avenues such as a case study on the use of our approach in the actual triage process of the considered open source projects and the actual developers' feedback (on arguably non-trivial tasks).

Our approach may not be universally applicable: We do not claim that our approach is universal. It might be possible that there are some commercial or legacy projects that lack the author information; however, we cannot categorically assert it. Our work shows that there are many open source projects with this information. An equivalent threat for such projects with the history-history based approaches is that the source code and/or bug history may not be available.

D. Reliability

Dataset not available: One of the main difficulties in conducting empirical studies is the access (or lack of it) to the dataset of interest. In our study, we used open source datasets that are publicly available. Also, we detailed the specifics of change requests that we used. The details of the bug and accuracy data for *ArgoUML*, *jEdit*, and *MuComamnder* are available at our online appendix (http://www.cs.wm.edu/semeru/data/icsm2012-authorship/).

Evaluation protocol not available: A concern could be that the lack of sufficient information on the evaluation procedure and protocol may limit the reproducibility of the study. We believe that our accuracy measures along with the evaluation procedure are sufficiently documented to enable replication on the same or even different datasets.

V. Related Work

McDonald and Ackerman [26] designed a tool coined as *Expertise Recommender* (ER) to locate developers with the desired expertise. The tool uses a heuristic that considers the most recent modification date when developers modified a specific module. In the case that multiple modules are considered, the developers that modified all the modules are considered. ER uses vector based similarity to identify technical support. Three query vectors (symptoms, customers, and modules) are constructed for each request. Subsequently, the vectors are compared to developer profiles. This approach has been designed for specific organizations and not tested on open source projects.

Minto and Murphy [28] developed a tool called *Emergent Expertise Locator* (EEL), which is based on the framework to compute coordination requirements between documents that was presented by Cataldo et al. [8]. EEL mines the history to determine how files were changed together and who committed those changes. Using this data, EEL suggests developers who can assist with a given problem. Another tool to identify developers with the desired expertise is *Expertise Browser* (ExB) [29]. The fundamental unit of experience is the Experience Atom (EA). The number of these EAs in a specific domain measures the developer experience. A code change that has been made on a specific file is the smallest EA.

Anvik and Murphy [2] conducted an empirical evaluation of two techniques for identifying expert developers. Developers acquire expertise as they work on specific parts of a system. They term this expertise as *implementation expertise*. Both techniques considered in the empirical evaluation are based on mining code and bug repositories. The first technique analyzes the check-in logs for modules that contain fixed source files. Developers who recently performed a change are selected and filtered. In the second technique, the bug reports from bug repositories are analyzed. The developers are identified from the CC lists, comments, and who fixed the bug. Their study concludes that both techniques have relative strengths in different ways. In the first technique, the most recent activity date is used to select developers.

Tamrawi et al. [38] used fuzzy-sets to the model bug-fixing expertise of developers based on the hypothesis that developers who recently fixed bugs are likely to fix them in the near future. Hence, only recent reports were considered to build the fuzzy-sets representing the membership of developers to technical terms in the reports. For incoming reports, developers are recommend by comparing their membership to the terms included in the new report.

A text based approach uses machine learning technique to automatically assign a bug report to a developer [1]. The resulting classifier analyzes the textual contents of a given report and recommends a list of developers with relevant expertise. *ExpertiseNet* also uses a text-based approach to build a graph model for expertise modeling [36]. Another recent approach to facilitate bug triaging uses graph based model based on Markov chains, which capture bug reassignment history [18]. Matter et al. [25] used the similarity of textual terms between a given bug report of interest and source code changes (*i.e.*, word frequencies of the *diff* given changes from source code repositories). A collection of past bug reports is not required by their approach. Likewise, our approach does not require the indexing of past bug reports.

There are a number of works on using MSR techniques to study and analyze developer contributions. Rahman and Devanbu [32] study the impact of authorship on code quality. They conclude that authors with specialized experience for a file is more important than general expertise. Bird et al. [7] perform a study on large commercial software systems to examine the relationship between code ownership and software quality. Their findings indicate that high levels of ownership are associated with less defects. A description of characteristics of the development team of *PostgreSQL* appears in a report by German [15]. His findings indicated that in the last years of *PostgreSQL* only two persons were responsible for most of the source code. Working time of open source software developers, based on email sent time, was analyzed by Tsunoda et al. [39]. Bird et al. [6] analyzed the communication and co-ordination activities of the participants by mining email archives. Del Rosso [11] built a social network of knowledge-intensive software developers based on collaborations and interaction. Ma et al. [24] proposed a technique that uses implementation expertise (*i.e.*, developers usage of API methods) to identify developers. Weissgerber et al. [41] depicts the relationship between the lifetime of the project and the number of files and the number of files each author updates by analyzing and visualizing the check-in information for open source projects. German [14] provided a visualization to show which developers tend to modify certain files by studying the modification records (MRs) of CVS logs. Fischer et al. [13] analyzed and related bug report data for tracking features in software.

VI. Conclusions

To the best of our knowledge, our approach is the only one to use a combination of a concept location technique and the source code authorship for assigning expert developers to change requests. It does not need to mine past change requests (*e.g.*, history of similar bug reports to resolve the bug request in question) or source code change repositories (e.g., commits to relevant source code to a change request). A single-version source code analysis of a system is only required. It expands the realm of available techniques to developer recommendation to include non-mining domains.

Our approach is perhaps simple and lightweight. Nonetheless, our empirical evaluation shows that it can be quite effective and competitive with the other approaches. For example, it is about 20% more accurate than an approach that uses machine learning on past bug reports in one system. Our empirical study did not show one technique outperforming the others across the board (not even the two previous techniques did so when compared with each other); however, we believe that our work could open up an interesting set of topics for future investigation. For example, triaging incoming change requests: bug or commit history, or code authorship, or all. When to use which approach and why? Further investigation would enable us to determine the exclusive and mutual benefits of these approaches.

ACKNOWLEDGMENT

We thank Bogdan Dit for collecting the execution information that was used in this paper to evaluate $xFinder_F$ and $Authorship_F$. We also thank Boyang Li for this help in the preliminary evaluation of a ML approach as a part of a CSci 635 project. This work is supported in part by NSF CCF-1156401, NSF CCF-1016868, NSF CCF- 1218129, and NSF CCF-0916260 grants. Any opinions, findings and conclusions expressed herein are those of the authors and do not necessarily reflect those of the sponsors.

REFERENCES

[1] Anvik, J., Hiew, L., and Murphy, G. C., "Who should fix this bug?" in Proceedings of 28th ICSE'06, pp. 361-370.

[2] Anvik, J. and Murphy, G., "Determining Implementation Expertise from Bug Reports", in Proceedings of MSR'07, Minneapolis, MN.

[3] Anvik, J. and Murphy, G. C., "Reducing the effort of bug report triage: Recommenders for development-oriented decisions", TOSEM'11, 20/3.

[4] Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., and Zimmermann, T., "What Makes a Good Bug Report?" in FSE'08.

[5] Bettenburg, N., Premraj, R., Zimmermann, T., and Kim, S., "Extracting Structural Information from Bug Reports", in Proc. of 5th MSR'08.

[6] Bird, C., Gourley, A., Devanbu, P., Gertz, M., and Swaminathan, A., "Mining Email Social Networks", in Proc. of MSR'06, pp. 137-143.

[7] Bird, C., Nagappan, N., Murphy, B., Gall, H., and Devanbu, P., "Don't Touch My Code! Examining the Effects of Ownership on Software Quality", in Proc. of ACM SIGSOFT ESEC/FSE'11.

[8] Cataldo, M., Wagstrom, P., Herbsleb, J., and Carley, K. M., "Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools", CSCW'06, pp.353-362.

[9] Collard, M. L., Kagdi, H. H., and Maletic, J. I., "An XML-Based Lightweight C++ Fact Extractor", in Proc. of IWPC'03, pp. 134-143.

[10] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R., "Indexing by Latent Semantic Analysis", Journal of the American Society for Info Science, vol. 41, no. 6, 1990, pp. 391-407.

[11] Del Rosso, C., "Comprehend and analyze knowledge networks to improve software evolution", JSM), vol. 21, no. 3, 2009, pp. 189-215.

[12] Dit, B., Revelle, M., Gethers, M., and Poshyvanyk, D., "Feature Location in Source Code: A Taxonomy and Survey", JSME, vol. doi: 10.1002/smr.567, 2012.

[13] Fischer, M., Pinzger, M., and Gall, H., "Populating a Release History Database from Version Control and Bug Tracking Systems", in Proc. of 19th ICSM'03, pp. 23-32.

[14] German, D. M., "An Empirical Study of Fine-grained Software Modifications", EMSE, vol. 11, no. 3, September 2006, pp. 369-393.

[15] German, D. M., "A Study of the Contributors of PostgreSQL", in Proc. of MSR '06, pp. 163 - 164.

[16] Gethers, M., Dit, B., Kagdi, H., and Poshyvanyk, D., "Integrated Impact Analysis for Managing Software Changes", in Proc. of ICSE'12.

[17] Hettmansperger, T. P., Statistical Inference Based on Ranks, 1st ed., John Wiley and Sons, 1984.

[18] Jeong, G., Kim, S., and Zimmermann, T., "Improving Bug Triage with Bug Tossing Graphs", in Proc. of 7th ESEC/FSE 2009.

[19] Kagdi, H., Gethers, M., Poshyvanyk, D., and Hammad, M., "Assigning Change Requests to Software Developers", JSME, vol. 24, no. 1, January 2012, pp. 3–33.

[20] Kagdi, H., Hammad, M., and Maletic, J. I., "Who Can Help Me with this Source Code Change?" in Proc. ICSM'08.

[21] Kagdi, H. and Poshyvanyk, D., "Who Can Help Me with this Change Request?" in Proc. ICPC'09, pp. 273-277.

[22] Leopold, E. and Kindermann, J., "Text Categorization with Support Vector Machines. How to Represent Texts in Input Space?" Machine Learning, vol. 46, no. 1, January 2002, pp. 423-444.

[23] Liu, D., Marcus, A., Poshyvanyk, D., and Rajlich, V., "Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace", in Proc. of 22nd IEEE/ACMASE'07, pp. 234-243.

[24] Ma, D., Schuler, D., Zimmermann, T., and Sillito, J., "Expertise Recommendation with Usage Expertise", in Proc. of 25th ICSM'09.

[25] Matter, D., Kuhn, A., and Nierstrasz, O., "Assigning Bug Reports using a Vocabulary-Based Expertise Model of Developers", MSR'09.

[26] McDonald, D., Ackerman, M., "Expertise Recommender: A Flexible Recommendation System and Architecture", in CSCW'00, pp. 231-240.

[27] McMillan, C., Linares-Vásquez, M., Poshyvanyk, D., Grechanik, M., "Categorizing Software Applications for Maintenance", in ICSM'11.

[28] Minto, S., Murphy, G., "Recommending Emergent Teams", MSR '07.

[29] Mockus, A. and Herbsleb, J., "Expertise Browser: a Quantitative Approach to Identifying Expertise", in Proc. 24th ICSE '02, pp. 503-512.

[30] Poshyvanyk, D., Guéhéneuc, Y. G., Marcus, A., Antoniol, G., and Rajlich, V., "Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval", IEEE TSE, vol. 33, no. 6, June 2007, pp. 420-432.

[31] Poshyvanyk, D. and Marcus, D., "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code", in Proc. of 15th IEEE ICPC'07, pp. 37-48.

[32] Rahman, F. and Devanbu, P., "Ownership, Experience and Defects: a Fine-Grained Study of Authorship", in ICSE'11, pp. 491-500.

[33] Revelle, M. and Poshyvanyk, D., "An Exploratory Study on Assessing Feature Location Techniques", in Proc. of 17th ICPC'09, pp. 218-222.

[34] Robles, G., González-Barahona, J., "Developer Identification Methods for Integrated Data from Various Sources", MSR'05, pp.106-110

[35] Runeson, P., Alexandersson, M., and Nyholm, O., "Detection of Duplicate Defect Reports Using Natural Language Processing", in Proc. of 29th IEEE/ACM ICSE'07, pp. 499-510.

[36] Song, X., Tseng, B., Lin, C., and Sun, M., "ExpertiseNet: Relational and Evolutionary Expert Modeling", in Proc. of UM'5.

[37] Surian, D., Liu, N., Lo, D., Tong, H., Lim, E. P., and Faloutsos, C., "Recommending People in Developers' Collaboration Network", in Proc. of 18th WCRE'11, pp. 379-388.

[38] Tamrawi, A., T. T. Nguyen, et al. (2011). Fuzzy Set and Cache-based Approach for Bug Triaging. 13th European conference on Foundations of software engineering (FSE'11), Szeged, Hungary.

[39] Tsunoda, M., Monden, A., Kakimoto, T., Kamei, Y., and Matsumoto, K.-i., "Analyzing OSS Developers' Working Time Using Mailing Lists Archives", in Proc. of MSR '06, pp. 181 - 182.

[40] Wang, X., Zhang, L., Xie, T., Anvik, J., and Sun, J., "An Approach to Detecting Duplicate Bug Reports using Natural Language and Execution Information", in Proc. of 30th ICSE'08, pp. 461-470.

[41] Weissgerber, P., Pohl, M., and Burch, M., "Visual Data Mining in Software Archives to Detect How Developers Work Together", MSR'07.