Amalgamating Source Code Authors, Maintainers, and Change Proneness to Triage Change Requests

Md Kamal Hossen, Huzefa Kagdi
Department of Electrical Engineering and Computer Science
Wichita State University
Wichita, KS 67260-0083
{mxhossen, huzefa.kagdi}@wichita.edu

Denys Poshyvanyk
Computer Science Department
The College of William and Mary
Williamsburg, VA 23185
denys@cs.wm.edu

ABSTRACT

The paper presents an approach, namely *iMacPro*, to recommend developers who are most likely to implement incoming change requests. *iMacPro* amalgamates the textual similarity between the given change request and source code, change proneness information, authors, and maintainers of a software system. Latent Semantic Indexing (LSI) and a lightweight analysis of source code, and its commits from the software repository, are used. The basic premise of *iMacPro* is that the authors and maintainers of the relevant source code, which is change prone, to a given change request are most likely to best assist with its resolution. *iMacPro* unifies these sources in a unique way to perform its task, which was not investigated and reported in the literature previously.

An empirical study on three open source systems, *ArgoUML*, *JabRef*, and *jEdit*, was conducted to assess the effectiveness of *iMacPro*. A number of change requests from these systems were used in the evaluated benchmark. Recall values for top one, five, and ten recommended developers are reported. Furthermore, a comparative study with a previous approach that uses the source-code authorship information for developer recommendation was performed. Results show that *iMacPro* could provide recall gains from 30% to 180% over its subjected competitor with statistical significance.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

Change Request, Expert Developer Recommendation, Software Maintenance, Triaging

1. INTRODUCTION

Software change requests and their resolution are an integral part of software maintenance and evolution. It is not uncommon in open source projects to receive tens of change requests daily that need to be promptly resolved [1]. Issue triage is a crucial activity in addressing change requests in an effective manner (e.g., within time, priority, and quality factors). The task of automatically assigning issues or change requests to the developer(s) who are most likely to resolve them has been studied under the umbrella of bug or issue triaging. A number of approaches to address this task have been presented in the literature [1, 16, 17, 28, 31]. They typically operate on the information available from software repositories, e.g., models trained from bug or issue repositories and/or source-code change repositories. Recently, an approach that does not require any mining of software repositories and only uses the source-code authorship information was introduced [17]; however, other sources of information, such as the change-proneness of source code, remain largely untapped in solving this problem.

We propose a new approach, namely *iMacPro*, for assigning the incoming change requests to appropriate developers who have necessary expertise for resolving them. *iMacPro* takes the textual description of an incoming change request (e.g., a short bug description) and locates relevant units (e.g., files) from a source-code snapshot. Latent Semantic Indexing (LSI), an information retrieval technique, is used in this step [10]. The relevant source-code units are ranked based on their change proneness, which is derived from their involvement in previous maintenance activities. Finally, the authors, extracted from a source code snapshot of these units, and the maintainers of these units, derived from past change activities, are forged together to arrive at the final list of developers. The developers in this list are ranked and are presumed to be best-fit candidates for resolving the change request in the order of their rank. *The basic premise of our approach is that the developers who are authors and/or maintainers of relevant source code, which is change prone, to a change request are most likely to best assist with its resolution.* In summary, our approach favors maintainers of change-prone source code. It uses LS**I**, **M**aintainers, **a**uthors, and **c**hange **Pro**neness of source code; hence, the name *iMacPro*. It needs the source code and its change history (i.e., commits); however, it does not require any training from the previously resolved bug reports.

To evaluate the accuracy of our technique, we conducted an empirical study on three open source systems/repositories: *ArgoUML*, *JabRef*, and *jEdit*. Recall metric values of the developer recommendations on a number of bug reports sampled from these systems are presented. That is, how effective our *iMacPro* approach is at recom-

ICPC 2014 Hyderabad, India

```
/*
 * Mode.java - jEdit editing mode
 * Copyright (C) 1998, 1999, 2000 Slava Pestov
 * Copyright (C) 1999 mike dillon

 * An edit mode defines specific settings for editing some type
 *
 * @author Slava Pestov
 * @version $Id: Mode.java 16022 2009-08-22 02:14:59Z daleanson
 */
public class Mode
{
    //{{{ Mode constructor
    /**
     * Creates a new edit mode.*/

}
```

Figure 1: An example file Mode.java from the open source system *jEdit*. The author Slava Pestov (spestov), Mike Dillon (mdillon) and (Dale Anson) daleanson are found in the header comment of the file which are all underlined in red.

mending the actual developer who ended up fixing these bugs. Additionally, our *iMacPro* approach is empirically compared with a previously reported approach that uses the source-code authorship information [17]. The results show that the proposed *iMacPro* approach outperformed the baseline competitor. Lowest recall gains of 28%, 56%, and 33% were recorded for *ArgoUML*, *JabRef*, and *jEdit* respectively. Highest recall gains of 31%, 57%, and 75% were recorded for *ArgoUML*, *JabRef*, and *jEdit* respectively.

Our paper makes the following noteworthy contributions in the context of recommending relevant developers to resolve incoming change requests:

1. To the best of our knowledge, our *iMacPro* approach is the first to integrate the change proneness, authors, and maintainers of relevant source code.

2. We performed a comparative study with another approach that makes an exclusive use of the source-code authorship information.

3. An empirical assessment of the contributions of the maintainer information toward the overall accuracy of *iMacPro*.

The rest of the paper is organized as follows: Our *iMacPro* approach is discussed in Section 2. The empirical study on three open-source projects and the results are presented in Section 3. Threats to validity are listed and analyzed in Section 4. Related work is discussed in Section 5. Finally, our conclusions and future work are stated in Section 6.

2. THE *iMacPro* APPROACH

Overall, our approach to assign incoming change requests to appropriate developers consists of the following steps:

1. We use Latent Semantic Indexing (LSI) [10] to locate relevant units of source code (e.g., files, classes, and methods) in a release of the software system that match the given textual description of a change request or reported issue. The indexed source-code release/snapshot is typically between the one in which an issue is reported and before the change request is implemented (e.g., a reported bug is fixed).

2. Source code units found from the above step are then ranked based on their change proneness. Change proneness of each source code entity is derived from its change history (elaborated in Sections 2.1 and 2.3).

3. The developers who authored and maintained these source code files are discovered and combined. Finally, a final ranked list of developers who are likely to best assist with the given change request is recommended.

Before we proceed with the finer details of our approach, key terms are defined and discussed next.

2.1 Key Terms and Definitions

`Author:` Authors of a source code entity (e.g., file) are the developers' identities found within it. Authors are typically found in the header comments of the source code entities (e.g., file, class, and method).

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. Tags such as *@author* and *@contributor* are commonly found in the header comments to denote the authorship information. Oftentimes, source-control systems automatically insert the tag *$Id* to signify an additional piece of developer information. Figure 1 shows that the authors *Slava Pestov*, *Mike Dillon*, and *daleanson* are found in the header comment of the file *mode.java*, see the underlined text in red in Figure 1. The extraction of authors from source code is discussed in Section 2.4.

`Maintainer:` Maintainers of a source entity (e.g., file) are the developers who performed changes on it (e.g., due to a bug fix or a feature implementation).

Maintainers are typically found in the commit information stored in a source code repository of a software system. Note that we differentiate between committers (developers who submitted the commits) and maintainers (developers who actually contributed the changes) whenever it is possible to do so. In situations when a developer identity is mentioned in the textual commit message, we consider them to be a maintainer and not the committer. Such scenarios do arise when someone else, other than the original developer who performed the actual change, is acting as a gatekeeper or facilitator. If no explicit developer is mentioned in the commit message, the committer is considered to be the maintainer.

Figure 2 shows a commit log from the open source system *jEdit*. This commit was performed by the developer *jarekczek*. The textual message in this log clearly states

```
<logentry revision="21981">
<author>jarekczek</author>
<date>2012-08-06T19:19:16.262992Z</date>
<paths>
<path prop-mods="false" text-mods="true" kind="file"
   action="M">/jEdit/trunk/org/gjt/sp/jedit/gui/AbstractContextOptionPane.java
</path>
<path prop-mods="false" text-mods="true" kind="file"
   action="M">/jEdit/trunk/org/gjt/sp/jedit/options/ContextOptionPane.java
</path>
</paths>
<msg>Tom Power fixed problems with his patch #3530786 - new option appearing
  in all context menu option panes, for example in VFS Browser.</msg>
</logentry>
```

Figure 2: An example of issue #3530786 from the open source system *jEdit*. jarekczek had submitted (committed) the changes but Tom Power is the actual developer who fixed the issue.

that the developer *Tom Power* fixed this issue by submitting a patch. Therefore, *Tom Power* and not *jarekczek* is the maintainer in this case. The extraction of maintainers from the source-code commits is discussed in Section 2.5.

`Issue Fixing Commit (IFC):` Issue Fixing Commits are the commits in a source-code repository that have explicit documentation of maintenance activities (e.g., corrective or adaptive). *IFC*s can be determined from the textual processing of the commit messages.

A common practice in the open source software development is for developers to include an explicit bug or issue *id* in the commit message. The presence of this information establishes the traceability between an issue or bug reported in the bug tracking system and the specific commit(s) performed to address it. Additionally, developers provide keywords such as *fix, gui, feature,* and *patch* in the commit messages to indicate a maintenance or evolutionary activity. A regular-expression based method can be employed to process commits and extract *IFC*s. The commit log shown in Figure 2 is an example of an IFC.

`Issue Change Proneness (ICP):` Issue Change Proneness of a source code entity is a measure of its change affinity as determined from Issue Fixing Commits (*IFC*s). A straightforward (yet as would be shown an effective) measure of the *ICP* of a source code entity e is given by the number of *IFC*s in the commit history that contain it.

$$ICP(e) =\mid \{\forall c \in IFCs \mid e \in c\} \mid \quad (1)$$

For example, the file **AbstractInputHandler.java** in *jEdit* was involved in a total of 11 commits from 2009-12-25 to 2006-10-02. Only two of these commits are *IFC*s. Therefore, the *ICP* of this source code file is 2.

2.2 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 [24]. This technique can be summarized in the following steps:

Creating a corpus from software: The source code is parsed using a developer-defined granularity level (i.e., files) and documents, i.e., in IR vocabulary, are extracted. A corpus is created, so that each file will have a corresponding document therein. Only identifiers and comments are extracted from the source code.

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 above steps are performed offline once, while the following two steps are repeated for a number of open change requests.

Using change requests: A set of words that describes the concept of interest constitutes the initial query, e.g., the short description of a bug or a feature described by the developer or reporter. This query is used as an input to rank the documents in the following step.

Relevant 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). We obtain a user-specified top n relevant documents. After these relevant documents are obtained, we treat them as a set of n documents and not a ranked list. The textual-similarity ranking of files is used for breaking ties in a later step (see Section 2.3).

We demonstrate the workings of the approach using an example from *ArgoUML*. The change request of interest here is the bug# 4563, which the reporter described as follows:

> *"Realization stereotype shows twice on abstraction"*

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 1,439 documents and 5,488 unique words. We formulated a search query using the bug's textual description. Table 1 shows the results of the search, i.e., files ranked in the order of their textual similarity scores.

Table 1: Top five files relevant to Bug#4563 in *ArgoUML*.

Rank	Files
1	diagram/ui/FigRealization.java
2	java/cognitive/critics/CrMultipleRealization.java
3	cognitive/critics/CrAlreadyRealizes.java
4	ui/foundation/core/PropPanelAbstraction.java
5	diagram/ui/FigAbstraction.java

2.3 Ranking Source Files with Issue Change Proneness

As discussed before, 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, ..., f_n for the bug b_i. We use the change-proneness measure to rank these top n files. *The rationale behind this choice is based on the premise that the larger the number of changes related to past requests (e.g., bug fixes) in which an LSI-relevant source code file is involved, the higher the likelihood of the same file requiring changes due to a given (new) change request.* For each relevant file, its *ICP* (see Section 2.1) is calculated. We consider the most recent m *IFC*s for each file in the computation of *ICP*. The parameter m is configurable. The n files are ranked based on their *ICP*s. The file with the highest *ICP* is ranked first, the one with the lowest *ICP* is ranked last, and so on. If multiple files have the same *ICP* value, their textual similarity values determine the ranks. At the conclusion of this step, the n relevant files are sorted.

The *ICP* values for each of the five files in Table 1 was computed for the purposes of ranking them based on their change proneness. We limited the computation to the most recent 20 *ICF*s for each file in this case. The column entitled *ICP* in Table 2 shows the corresponding value of each file. These files are presented in the rank of their *ICP*. Clearly, this ranking differs from the LSI ranking in Table 1. The files *FigRealization.java* and *CrMultipleRealization.java* have the same *ICP* value of 4. The file *FigRealization.java* is ranked higher than the file *CrMultipleRealization.java* based on the LSI similarity (rank) in Table 1, i.e., it is ranked ahead.

2.4 Extracting Authors from Source Code

The next step is to extract authors from each of the top n relevant files obtained from Step 2.2. Specifics of this extraction component are provided below:

Obtaining source code files: The source code of each of the top relevant files that are retrieved by the concept location component of our technique is first obtained. These source code files are derived from a system snapshot between when the change request is reported and before it is resolved.

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; however, any lightweight source-code analysis methods, including regular expressions or island grammar [23], can be also used.

Extracting header comments: All the header comments are extracted from each srcML file. The header comments are generally the first comments in source code files, source code classes, and/or methods.

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 of the relevant files. Note that the same developer could have multiple identities. We extracted all the entities of each developer from the project resources, and mapped them to a unique identifier. For example, the identities $MichielvanderWulp$ (full name), *mvw@tigris.org* (email address) and *mvw* (user name) represent the same developer, which is mapped to the identity *mvw*. Similarly, the identities *jaap.branderhorst@xs4all.nl* and *jaap* represent the same developer, which is mapped to the identity *jaap*.

Ranking Authors: For each file, the authors are ranked according to the lexical positions of the constructs in which they are found. That is, a top-down, left-right order is followed. For example, the authors appearing in the header comment of the file are ranked higher than those appearing in the header comment of the (main) class. If multiple authors appear in the same comment, the one that is encountered first lexically is ranked first, and so forth. It is possible that the same author is discovered from multiple places in the same file. We assign the rank of the earliest lexical position to such an author. Figure 1 shows that the author *SlavaPestov* is found in the first line of the header comment of the file *Mode.java*. Thus, the author *SlavaPestov* is ranked first. The same author is discovered again in the header comments of the class *Mode*, which is ignored. Next the author *Mike Dillon* is found in the copyright header in the lexical order. Finally, the author *Dale Anson* is found. The final ranked list of authors for the file *mode.java* is [*spestov*, *mdillon*, *daleanson*]. After this step, a ranked author list for each of the top n relevant files is established.

In our running example, the source code of each of the five files in Table 1 was then processed to find a ranked list of authors. Table 2 shows this list for each file in the column entitled *Authors*. The file *FigRealization.java* has only the author $tfmorris$. The file *CrAlreadyRealizes.java* has the ranked list of authors [$linus$, $jrobbins$].

2.5 Extracting Maintainers from Change History

For each relevant file, its most recent *IFCs* are sorted with the most recent commit appearing first and the least recent commit appearing last. Maintainers from these *IFCs* are extracted (see Section 2.1). We compiled a list of developer IDs from the software repositories and project documentation, similar to extracting authors. The maintainer of the most recent commit is ranked first and that of the least recent commit is ranked last. *The rationale for this choice is based on the premise that developers who made the most recent changes are likely to be most familiar with the current state of source code. Therefore, they would be better able to assist with a given change request than others.* In cases where the same maintainer was responsible for multiple commits in *IFCs*, the highest ranked position is retained and others are discarded. After this step, a ranked maintainer list for each of the top n relevant files is established.

The column entitled *Maintainers* in Table 2 shows the ranked list of maintainers for each of the files. The file *PropPanelAbstraction.java* has the ranked list of maintainers *[mvw, linus, mkl, kataka, 1sturm]*. The developer *mvw* was the maintainer of the most recent *IFC*, whereas, the maintainer *1strum* was the maintainer of the least recent, i.e., oldest, one. Although file *PropPanelAbstraction.java* has the *ICP* value of 7, it has only 5 maintainers. Each commit typically has a single maintainer (one committer, for sure, unless an anonymous commit was a result of a migration process from an automatic tool). Therefore, in this case, there was at least one maintainer who performed multiple *IFC*s on this file. Only the highest ranked position, i.e., the most recent *IFC*, of such as a maintainer is preserved.

2.6 Fusion of Authors and Maintainers of Change-Prone Source Code to Recommend Developers

We now describe the details of combining the change proneness, authors, and maintainers of source code, relevant to a given bug, to recommend the final ranked-list of developers. From Section 2.4, 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 have three authors. Although, the ranked list of authors of a single file does not have any duplication, two files may have common authors. The final ranked lists of authors for the top n relevant files ranked based on their *ICP* values are given by the matrix D_a below:

$$D_a = \begin{pmatrix} f_1 & D_{af_1} \\ f_2 & D_{af_2} \\ .. & .. \\ .. & .. \\ f_n & D_{af_n} \end{pmatrix} \quad D_{af_i} = [\, a_1 \; a_2 \; \cdots \; a_l \,] \tag{2}$$

In Equation 2, D_{af_i} represents the ranked list of authors, with no duplication, for the file f_i. a_j is the j^{th} ranked author in the file f_i, which contains l unique authors. The ranks for the authors are in the range $[1, l]$.

From Section 2.5, there is a one-to-many relationship between the source code file and maintainers. Each file f_i may have multiple maintainers; however, it is not necessary for all the files to have the same number of maintainers. Although, the ranked list of maintainers of a single file does not have any duplication, two files may have common main-

tainers. The final ranked lists of maintainers for the top n relevant files ranked based on their *ICP* values are given by the matrix D_m below:

$$D_m = \begin{pmatrix} f_1 & D_{mf_1} \\ f_2 & D_{mf_2} \\ .. & .. \\ .. & .. \\ f_n & D_{mf_n} \end{pmatrix} D_{mf_i} = [\, m_1 \; m_2 \; ... \; m_o \,] \tag{3}$$

In Equation 3, D_{mf_i} represents the ranked list of maintainers, with no duplication, for the file f_i. m_j is the j^{ih} ranked maintainer in the file f_i, which contains o unique maintainers. The ranks for the maintainers are in the range $[1, o]$.

To obtain a combined ranked-list of developers for each file f_i, i.e., D_{ma}, ranked lists of maintainers (D_m) and authors(D_a) are assembled.

$$\begin{aligned} D_{ma} &= D_m \uplus D_a \\ &= \begin{pmatrix} D_{maf_1}=D_{mf_1} \uplus D_{af_1} \\ D_{maf_2}=D_{mf_2} \uplus D_{af_2} \\ .. \\ .. \\ D_{maf_n}=D_{mf_n} \uplus D_{af_n} \end{pmatrix} \\ &= \begin{pmatrix} f_1 \; [\, d_1 \; d_2 \; ... \; d_{max(l,o)} \,] \\ f_2 \; [\, d_1 \; d_2 \; ... \; d_{max(l,o)} \,] \\ .. \\ .. \\ f_n \; [\, d_1 \; d_2 \; ... \; d_{max(l,o)} \,] \end{pmatrix} \end{aligned} \tag{4}$$

We employ a round-robin merging algorithm. For each file f_i, the first position d_1 on the list is occupied by the highest-ranked m_1 maintainer, i.e., the maintainer appearing first in the maintainer list. For the second position d_2 on this combined list, the highest-ranked author a_1, i.e., the author appearing first in the author list is considered. *The rationale behind picking first from the maintainer list and then from the author list is based on the premise that the maintainer who contributed recent changes to a file is more likely to have relevant knowledge than its authors.*

For each file f_i, we eliminated redundancies within the individual author and maintainer lists (D_{mf_i}) and (D_{af_i}); however, these two lists may have developers in common, i.e., the maintainer and author are the same developer. Therefore, if a developer is already in the combined list of developers D_{maf_i}, it is discarded and the next one is picked from the author or maintainer list depending on where the redundancy was found. For example, if it was the author list's turn to pick a developer for the j^{th} position and that developer is already in the combined list, the next one on the maintainer list is considered for this position. If either of the author or maintainer list is exhausted, the remaining balance is fulfilled by the other list, barring no further redundancy nor is this list also exhausted. At the conclusion of this step, we have a ranked-list of developers for each file.

In Table 2, the ranked-list of combined developers, i.e., D_{ma}, for each file is shown in the last column (**Combined Developers(D_{ma})**). For example, the ranked-lists of maintainers (D_m) and authors (D_a) for the file *CrAlreadyRealizes.java* are $[mkl, mvw, linus, kataka]$ and $[linus, jrobbins]$. According to our round-robin method of combining, we pick *mkl* from the maintainer list first, *linux* from the author list second, *mvw* from the maintainer list third, and *jrobbins* from the author list fourth. *linus* from the maintainer list is discarded, as it already appears in the combined list. Because the author list is exhausted, we simply append *kataka* to the combined list. Therefore, the combined list D_{ma} for the file *CrAlreadyRealizes.java* is $[mkl, linus, mvw, jrobbins, kataka]$.

The combined lists of developers in Equation 4 contain ranked developers for each file; however, we need to recommend a user specified k developers. Therefore, we need to obtain the absolute ranking of developers from D_{ma}. To do so, we coalesce developers from D_{ma} into a single ranked list of candidate developers. We start with the highest-ranked developer for the highest-ranked file in D_{ma}, move on to the highest-ranked developer for the second-highest ranked file, and so on. That is, the highest-ranked developers of all the files are first added. Once they are added, the second-highest ranked developers of the files (traversed by their rank order) are added. This process continues, until the lowest-ranked developers from all the lowest-ranked files are merged into the final ranked-list of developers. Once again, the elimination of developer redundancy occurring in multiple files is handled in the same way used for generating D_{ma}. That is, we have an order-preserving union of developers from the traversal of ranked files and their developers in D_{ma}. At the conclusion of this step, we have a ranked-list of developers for the given change request, i.e., D_f in Equation 5.

$$D_f = d \mid \uplus_{i=1}^{n} \forall d \in D_{maf_i} \in D_{ma} \tag{5}$$

The top k developers recommended to address the given change request are the top k developers in D_f. This step concludes our *iMacPro* approach.

In our running example, the formation of the combined developer list, i.e., D_f starts by taking the first developer from the D_{ma} of the highest ranked file. That is, *mvw* from the file *PropPanelAbstraction.java*. Next, the first developer from the D_{ma} of the second-highest ranked file is considered and retained. That is, *mkl* from the file *CrAlreadyRealizes.java*. Continuing in this fashion, *mkl* from the file *FigRealization.java* is considered; however, it is already in the list of final developers. Therefore, it is discarded. Similarly, *mkl* and *mvw* from the files *CrMultipleRealization.java* and *FigAbstraction.java* are eliminated, as they were already picked before. At this point, all the first position developers of all the files are exhausted and we have developers *mvw* and *mkl* on the combined list of developers, D_f. Table 3 details the formation of the final list of combined developers. The column **Position** shows the workings of the i^{th} position developer in the D_{ma} of each file. The row with the position value 2 shows the second highest developers considered from each file in their ranked order. After all the positions are considered, the final list of combined developers, D_f, for *bug#*4563 is *[mvw,mkl, bobtarling, linus, tfmorris, agauthie, jrobbins, kataka]* (see the row labeled D_f. Finally, for a user specified cutoff of $k = 5$, the recommended ranked list of developers would be **[mvw, mkl, bobtarling, linus, tfmorris]** (see the row labeled $D_{f@k=5}$).

On examining the source code repository of *ArgoUML*, we found that *mvw* was the developer who resolved the *bug#*4563 in *commit#*11821. As can be seen in the final recommendation list $D_{f@k=5}$, *mvw* is the first ranked developer. Therefore, our *iMacPro* approach would have recommended the correct developer who resolved this bug with only one recommendation.

3. EMPIRICAL STUDY

The main purpose of this case study was to investigate

Table 2: The authors and maintainers extracted from each of the top five files relevant to Bug# 4563.

Rank	Files	ICP	Ranked Maintainers(D_m)	Ranked Authors(D_a)	Combined Developers(D_{ma})
1	.../PropPanelAbstraction.java	7	mvw, linus, mkl, kataka, 1sturm	bobtarling	mvw, bobtarling, linus, mkl, kataka, 1sturm
2	.../CrAlreadyRealizes.java	6	mkl, mvw, linus, kataka	linus, jrobbins	mkl, linus, mvw, jrobbins, kataka
3	.../FigRealization.java	4	mkl, linus, kataka	tfmorris	mkl, tfmorris, linus, kataka
4	.../CrMultipleRealization.java	4	mkl, linus	mvw, jrobbins	mkl, mvw, linus, jrobbins
5	.../FigAbstraction.java	1	mvw	tfmorris, agauthie	mvw, tfmorris, agauthie

Table 3: Developers taken from individual files in positions 1 to 5 from Table 2. The strikeout developers are discarded and others are retained.

Position	Combined Developers(D_f)
1	mvw, mkl, ~~mkl~~, ~~mkl~~, ~~mvw~~
2	bobtarling, linus, tfmorris, ~~mvw~~, ~~tfmorris~~
3	~~linus~~, ~~mvw~~, sout linus, ~~linus~~,agauthie
4	~~mkl~~, jrobbins, kataka ,~~jrobbins~~
5	~~kataka~~, ~~kataka~~
D_f	mvw, mkl, bobtarling, linus, tfmorris, agauthie, jrobbins, kataka
$D_{f@k=5}$	mvw, mkl, bobtarling, linus, tfmorris

how well our *iMacPro* approach recommends expert developers to assist with incoming change requests. Moreover, we compare *iMacPro* with a previous approach for developer recommendation (denoted here as *iA* [17]). Similar to *iMacPro*, *iA* uses LSI to identify relevant source files and then uses the author information in those files for recommending developers. However, it does not use the maintainer information from the source code commit history. Another way *iMacPro* differs from *iA* is in the use of the *change-proneness*, i.e., *ICP* in ranking the relevant files. Therefore, we provide another comparison between *iMacPro* and an approach that is identical to *iMacPro* but does not use the maintainer information (denoted here as *iAcPro*). This comparison would permit the assessment of the impact of including maintainers in *iMacPro*.

We investigate the following research questions (RQ_s) in our case study:

- **RQ_1:** What is the accuracy of *iMacPro* when assessed on open-source systems?
- **RQ_2:** How does the accuracy of *iMacPro* compare to *iA*, i.e., a previous approach based on the source code authorship alone?
- **RQ_3:** How does the accuracy of *iMacPro* compare to *iAcPro*, i.e., when the maintainer information is not utilized in *iMacPro*, giving us *iAcPro*?

The rationale behind RQ_s is two-fold: 1) To assess whether our *iMacPro* approach can identify correct developers to handle change requests in open source systems. 2) To discover how well the accuracy of the *iMacPro* approach compares to those of *iA* and *iAcPro* approaches?

3.1 iA and iAcPro Approaches

We briefly summarize the *iA* and *iAcPro* approaches for reproducibility purposes. The *iA* consists of two steps:

1. The first step is identical to the *iMacPro* approach (see Section 2.2).
2. A union of all the authors appearing in all n files from the previous step is created, which gives a set of d unique authors. For each author d_i, the number of files in which they appear is counted. Once this frequency count for each author is obtained, all the authors are sorted in descending order of their file frequency counts. From this sorted list of authors, the top k ranked authors to assist with fixing the bug/change request in question are recommended.

We chose this approach for a comparative baseline for accuracy here because it was shown to perform comparably or better than a few of its contemporaries [17]. This previous comparison of *iA* included a machine learning approach applied on past bug reports to recommend developers.

The *iAcPro* approach is identical to *iMacPro*; however, maintainers are not included in the list of recommended developers. After getting the list of n relevant files from IR, a list of top k developer is created using the same ranking mechanism of *iMacPro*. In *iAcPro*, source files are sorted based on their *ICP* values, similar to *iMacPro*; however, only the authors of the files are considered.

The accuracy comparison between the aproaches *iMacPro* and *iAcPro* would help us assess the level of the maintainer contribution to the accuracy of *iMacPro*. That is, *is it really worthwhile to put in the additional work of extracting maintainers to recommend developers, or would the author and change proneness information would suffice?*

3.2 Subject Software Systems

The context of our study is characterized by three open source Java systems: *jEdit* v4.3, a popular text editor, *ArgoUML* v0.22, a well-known UML modeling tool, and *JabRef* v1.8, an open source bibliography reference manager. *ArgoUML* and *jEdit* were used in a previous study [17]. The sizes of these considered systems range from 75K to 150K LOC and contain between 4K and 11K methods. The descriptive statistics of these systems are given in Table 4.

3.3 Building The Benchmarks

For each of the subjected systems, we created a benchmark of bugs and the actual developers who fixed them to

Table 4: Subject software system used in 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
JabRef	1.8	38,680	311	2,465	2,464

conduct our case study. 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, 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., gold set, which forms our actual developer set for evaluation. A vast majority of change requests are handled by a single developer (i.e., commit contributors). In cases where we found the committer to be different from the maintainer who contributed the change, the maintainer was considered to be the one who resolved the bug. The change requests in the benchmark include bug fixes, feature requests, and feature enhancements.

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

3.4 Metrics and Statistical Analyses

We evaluated the accuracy of each one of the approaches, for all the reports in our testing set, using the recall metrics used in previous work [1, 17, 31]. For a b number of bugs in the benchmark of a system and a k number of recommended developers, the formula for the recall@k is given below:

$$recall@k = \frac{1}{b}\frac{\sum_{i=1}^{b} \mid RD(b_i) \cap AD(b_i) \mid}{\mid AD(b_i) \mid} \quad (6)$$

where $RD(b_i)$ and $AD(b_i)$ are the recommended developer by the approach and the actual developer who resolved the issue for the bug b_i.

These metrics were computed for recommendation lists of developers with different sizes, i.e., $k = 1$, $k = 5$, and $k = 10$ developers. The reason for not using another popular metric precision is that a change request (or bug fix) typically has one developer implementing it, i.e., $|AD(b_i)| = 1$. Therefore, for $k = 1$ to 10, there is typically only one correct answer and others are incorrect. Therefore, the best precision values would range from 1.0 to 0.1.

We applied the One Way ANOVA test to validate whether there was a statistically significant difference with $\alpha = 0.05$ between the results. We used this non-parametric test because we did not assume normality in the distributions of 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 (the alternative hypotheses could be easily derived from the respective null hypotheses):

- **H-1:** There is no statistically significant difference between the recall@k values of *iMacPro* and *iA*.
- **H-2:** There is no statistically significant difference between the recall@k values of *iMacPro* and *iAcPro*.

3.5 Results

For each change request of each subject system in the benchmark, we parametrized our *iMacPro* approach to recommend top *one*, top *five*, and top *ten* developers. We considered top *five* relevant files from the LSI-based approach. The source code snapshots used for extracting authors were taken from when or before the bugs were reported. The source code commits used for the change proneness measurement and extracting maintainers were from the instances before the issues were resolved. That is, there was no instance where authors and maintainers were extracted after the given issue was already resolved (which would have been a fault in the experiment design). These recommendations were compared with the actual developer who resolved the considered change request to compute the recall value. The recall@1, recall@5, and recall@10 values for each system were calculated (see Equation 6). Similarly, the recall@1, recall@5, and recall@10 values for each system were calculated for the two competing approaches *iA* and *iAcPro*.

Table 5 shows the recall@k values for all the three approaches. As expected, the recall value generally increases with the increase in the k value for each approach. For example, the recall@1, recall@5, and recall@10 values of *iMacPro* on *ArgoUML* are 0.15, 0.54, and 0.58 respectively. That is, *iMacPro* was able to recommend the correct developer for 15%, 54%, and 58% of change requests in the *ArgoUML* benchmark by recommending one, five, and ten developers. Table 5 suggests that the recall values were about to plateau at $k = 10$ for all three approaches. Therefore, it was not necessary to go beyond $k = 10$. The **Recall@k** column in Table 5 shows recall values of the three approaches.

To answer the research question RQ_1, our approach *iMacPro* reported recall values ranging from 0.15 to 0.69 on three open source projects. Therefore, we posit that it can perform well when subjected to real world open source systems. To answer the research question RQ_2, we compared the recall values of *iMacPro* and *iA* for $k = 1$, $k = 5$, and $k = 10$. That is, we computed the recall gain of *iMacPro* over *iA*, which is computed using the formula:

$$gain@k_{iMacPro-iA} = \frac{recall@k_{iMacPro} - recall@k_{iA}}{recall@k_{iA}} \times 100 \quad (7)$$

The ***iMacPro* gain over *iA* %** column in Table 5 shows the recall gains of *iMacPro* over *iA* for the different k values. As can be seen, *iMacPro* clearly outperforms *iA* in the cases of *jEdit* and *JabRef* for all the k values. The gains in these two systems range from 33% to 75%. There was a mixed report from *ArgoUML*: *iA* performed better than *iMacPro* for $k = 1$ (a negative gain of 23%), whereas, *iMacPro* performed better than *iA* for $k = 5$ and $k = 10$ (positive gains of 27% and 30%). **In summary, the overall results suggest that *iMacPro* would generally**

Table 5: Recall@1, 5, and 10 of the approaches *iA*, *iAcPro*, and *iMacPro* measured on the *ArgoUML*, *jEdit* and *JabRef* benchmarks.

System/Benchmark	Top k	Recall@k			*iMacPro* gain over *iA* %	*iMacPro* gain over *iAcPro* %
		iA	*iAcPro*	*iMacPro*		
ArgoUML	1	0.19	0.18	0.15	-23.09	-13.37
82	5	0.39	0.39	0.54	28.27	39.41
Change Requests	10	0.40	0.40	0.58	30.62	44.13
jEdit	1	0.04	0.08	0.15	74.90	99.48
52	5	0.31	0.35	0.46	33.33	33.30
Change Requests	10	0.35	0.35	0.54	35.71	55.55
JabRef	1	0.17	0.14	0.39	57.14	179.99
36	5	0.31	0.31	0.69	55.99	127.23
Change Requests	10	0.31	0.31	0.69	55.99	127.23

perform better than *iA* in terms of recall. Augmenting the authorship-based approach with the change-proneness and maintainer information typically leads to improvements in accuracy.

To answer the research question RQ_3, we compared the recall values of *iMacPro* and *iAcPro* for $k = 1$, $k = 5$, and $k = 10$. That is, we computed the recall gain of *iMacPro* over *iAcPro*, which is computed using the formula:

$$gain@k_{iMacPro-iAcPro} = \frac{recall@k_{iMacPro} - recall@k_{iAcPro}}{recall@k_{iAcPro}} \times 100 \quad (8)$$

The ***iMacPro* gain over *iAcPro* %** column in Table 5 shows the recall gains of *iMacPro* over *iAcPro* for the different k values. As can be seen, *iMacPro* clearly outperforms *iAcPro* in the cases of *jEdit* and *JabRef* for all the k values. The gains in these two systems range from 33% to 180%. There was a mixed report from *ArgoUML*: *iAcPro* performed better than *iMacPro*for $k = 1$ (a negative gain of 13%), whereas, *iMacPro* performed better than *iAcPro*for $k = 5$ and $k = 10$ (positive gains of 39% and 44%). **In summary, the overall results suggest that *iMacPro* would generally perform better than *iAcPro* in terms of recall. The maintainer component is a substantial contributor to the effectiveness (accuracy) of *iMacPro***.

To test the hypothesis H_1, we applied the One Way ANOVA test on the recall values of *iMacPro* and *iA* for each of the change request in the benchmark of each subject system. The ***iMacPro-iA*** column in Table 6 shows the p-values for $k = 1$, $k = 5$, and $k = 10$ for each subject system. In the cases of *jEdit* and *JabRef*, the p-values are $\leq$0.05, so we can reject the null hypothesis H_1. In the case of *ArgoUML*, the p-values are $\leq$0.05 for $k = 5$ and $k = 10$, so we can reject the null hypothesis H_1. Note that in these cases, the reported gains were positive in Table 5. In case of *ArgoUML*, the p-value is >0.05 for $k = 1$, so we cannot reject the null hypothesis H_1. Revisiting the corresponding recall@1 for this case in Table 5, *iA* performed better than *iMacPro*; however, this observation is not statistically significant. The only case in which *iMacPro* seemed to have a disadvantage over *iA* is not statistically valid. One of the reasons that could be attributed for $k = 1$ is that *iA* was able to recommend only the resolutions performed by one developer correctly, whereas, *iMacPro* was able to do so for multiple developers. That is, *iA* performed much better for the issues in the benchmark for this one specific developer than *iMacPro*; however *iMacPro* had an advantage for issues resolved by other developers. Therefore, there is no clear winner for $k = 1$ statistically in *ArgoUML*. Eventually, the diversity of recommended developers by *iMacPro* was advantages over *iA*, which can also be seen in the results for $k = 5$, and $k = 10$. **In summary, we reject the null hypothesis H_1 in favor of *iMacPro* over *iA***.

To test the hypothesis H_2, we applied the One Way ANOVA test on the recall values of *iMacPro* and *iAcPro* for each of the change request in the benchmark of each subject system. The ***iMacPro-iAcPro*** column in Table 6 shows the p-values for $k = 1$, $k = 5$, and $k = 10$ for each subject system. Similar to the ***iMacPro-iA*** comparison, the p-values support rejection of the null hypothesis H_2 in all but two cases. These two cases are for $k = 1$ for *ArgoUML* and *jEdit*. In these two cases, neither *iMacPro* nor *iAcPro* has an advantage over the other. One of the reasons that could be attributed for this observation is that both authors and maintainers were the same developers. Therefore, neither of these approaches offered a specific competitive edge. **In summary, we reject the null hypothesis H_2 in favor of *iMacPro* over *iAcPro***.

Now, we provide representative bugs from the subject systems detailing *iMacPro*'s performance compared to the two other approaches. For example, *iMacPro* recommends the correct developer (*coezbek*) in the 1^{st} position for *bug*#1548875 in *jEdit*, whereas, *iA* and *iAcPro* failed to recommend the correct developer at all. In *jEdit* system for *bug*#2946041 *iMacPro* was able to find the correct developer (*kpouer*) in the 1^{st} position, whereas, *iA* and *iAcPro* found him in the the 2^{nd} position. In *ArgoUML*, *iA* and *iAcPro* recommended the correct developer ($tfmorris$) for bug#4720 in the 5^{ith} and 3^{rd} positions respectively, whereas, *iMacPro* found it in the 1^{st} position.

4. THREATS TO VALIDITY

We identified threats to validity that could influence the results of our study and limit their generalization.

4.1 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?

Accuracy measures and correctness of developer recommendations: We used two widely used metrics recall and recall gain in our study. We considered a gold-set to be

Table 6: p-values from applying One Way ANOVA on recall@k values for each subject system.

System	Top k	p-value	
		iMacPro-iA	*iMacPro- iAcPro*
ArgoUML	1	≤0.54	≤0.68
	5	≤0.05	≤0.05
	10	≤0.03	≤0.03
jEdit	1	≤0.05	≤0.11
	5	≤0.03	≤0.04
	10	≤0.01	≤0.01
JabRef	1	≤0.04	≤0.02
	5	≡0.00	≡0.00
	10	≡0.00	≡0.00

developers who contributed source code changes to address change requests. Of course, it is possible that other team members are also equally qualified to handle these change requests; however, such a gold-set would be very difficult to ascertain in practice without involving the project stakeholders, for example. Moreover, these project stakeholders would need to remember exactly who were good alternative developers at that time. Thus, we hypothesize that building such datasets by interviewing project managers could be an error-prone activity with substantial bias. Nonetheless, our undertaken benchmark provides careful accuracy values yet perhaps conservative bounds.

LSI-based matching of change requests to relevant files: The IR-based concept location tool based on LSI does not always return the classes (files) that are found in the commits related to the bug fixes or change request implementations in all the cases. However, based on our prior work we observed that the files that were recommended as textually similar 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).

Measuring change-proneness of source code files: Although we understand that it is possible to use other metrics for measuring change proneness, we decided to use issue fixing commits as a measure of the source-code file change affinity. Our rationale is based on the fact that it is a common practice in the open source development to include explicit issues IDs in the commit messages, which can be captured and counted effectively using a very lightweight approach.

4.2 Internal Validity

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

Missing Traceability. We only considered commits with the explicit documentation of maintenance activities, which were determined from keyword matching. It should be noted that it is a common approach used in a number of previous approaches and studies. Nonetheless, we do not claim that our approach is exhaustive in extracting all the issue-fixing commits for all the change requests. Bachmann et al. [3] identified the missing traceability between bug reports and commits in *Apache*. Wu et al. [30] proposed a machine-learning approach to identify such missing links. In the future, we plan to incorporate this element in *iMacPro*.

Merging of developers from authors and maintainers. When authors and maintainers are combined in *iMacPro* to obtain the final list of combined developers, maintainers are picked first. Although our empirical study shows that this choice worked very well, it is possible that a different selection scheme (e.g., authors first) could produce a different (perhaps better) performance. We plan to investigate this topic in future studies.

Ranking of source code files based on change-proneness alone. We ranked the relevant source code units to the given change request based on their change proneness alone; however, it is possible that another ranking mechanism could have impact on the performance. We plan to examine the impact of a ranking mechanism based on a combination of the textual similarity and change-proneness measures.

Developer identity mismatch. Although we carefully examined all the available sources of information to match different identities of the same developer, it is possible that we missed or mismatched a few cases.

Impact of other factors: We demonstrated a positive relationship between the developers recommended with *iMacPro* and the developers who fixed them (i.e., our constructed benchmark). It is possible that other factors, such as schedule, work habits, technology fade or expertise, and project policy/roles may also influence the triaging results. A definitive answer to this question would require another set of studies, which we believe is beyond the scope of this work.

4.3 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: We evaluated three open source systems, which we believe are good representatives of large-scale, collaboratively developed software systems. However, we cannot claim that these results would equally hold on other systems (e.g., closed source).

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 the developer recommendation problem. The approach of "the 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.

4.4 Reliability

We discuss threats that could impact replication of our evaluation study.

Dataset not available: One of the main difficulties in conducting empirical studies is the access (or lack of it) to the dataset of interest. We used open source datasets that are publicly available. The details of the bug and accuracy data for *ArgoUML*, *JabRef*, and *jEdit* are available at our online appendix `http://serl.cs.wichita.edu/svn/projects/dev-rec-authors/trunk/Paper/icpc2014-appendix/`.

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 on different datasets.

5. RELATED WORK

McDonald and Ackerman [20] 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. 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 [21] developed a tool called Emergent Expertise Locator (EEL), which is based on the framework of Cataldo et al. [8] to compute coordination requirements between documents. 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) [22]. 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 bug fixes. 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. [28] 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.

An approach uses a 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 [27]. Another approach to facilitate bug triaging uses a graph model based on Markov chains, which captures the bug reassignment history [15]. Matter et al. [19] 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).

There are a number of works on using MSR techniques to study and analyze developer contributions. Rahman and Devanbu [25] 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. [6] 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 fewer defects. A description of characteristics of the development team of PostgreSQL appears in a report by German [14]. His findings indicated that in the last years of PostgreSQL only two persons were responsible for most of the source code. Bird et al. [5] 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. [18] proposed a technique that uses implementation expertise (i.e., developers usage of API methods) to identify developers. Weissgerber et al. [29] 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 [13] provided a visualization to show which developers tend to modify certain files by studying the modification records (MRs) of CVS logs. Fischer et al. [12] analyzed and related bug report data for tracking features in software. Bortis et al. [7] introduced PorchLight a tag-based interface and customized query expression to offer triagers the ability to explore, work with, and assign bugs in groups. Shokripour et al. [26] proposed an approach for bug report assignment based on predicted location (source code) of the bug and showed advantages of this approach over activity based approach. Corley et al. [9] built *ohm* tool that used a combination of software repository mining and topic modeling for measuring the ownership of linguistic topics in source code. Begel et al. [4] conducted a survey of inter-team coordination needs and presented a flexible *Codebook* framework that can address most of those needs.

6. CONCLUSIONS AND FUTURE WORK

We presented the *iMacPro* approach to recommend developers who are most likely to implement incoming change requests. *iMacPro* determines and integrates, authors and maintainers of relevant source code files, which are change prone, to a given change request. Such a combined approach to recommend developers was not investigated and reported in the literature previously. Moreover, an empirical study on three open source systems showed that *iMacPro* can outperform a previous approach with statistically significant recall gains. The results also justify the accuracy benefit of including source-code maintainers in the functioning of *iMacPro*.

In the future, we plan to conduct additional empirical studies to further validate *iMacPro*. Furthermore, we will investigate other sources of information that could further improve its effectiveness. These sources include different measures for change proneness, ranking of relevant source code, and merging schemes for authors and developers.

7. ACKNOWLEDGMENTS

We would like to thank Bogdan Dit from the College of William and Mary for his help in verifying the data and obtaining IR-based results. This work is supported in part by the NSF CCF-1156401 and NSF CCF-1016868 grants. Any opinions, findings and conclusions expressed herein are those of the authors and do not necessarily reflect those of the sponsors.

8. REFERENCES

[1] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th ACM International Conference on Software Engineering*, ICSE '06, pages 361–370, New York, NY, USA, 2006.

[2] J. Anvik and G. Murphy. Determining implementation expertise from bug reports. In *Fourth International Workshop on Mining Software Repositories (MSR), 2007 ICSE Workshops MSR '07*, pages 2–2, 2007.

[3] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein. The missing links: Bugs and bug-fix commits. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 97–106, New York, NY, USA, 2010. ACM.

[4] A. Begel, Y. P. Khoo, and T. Zimmermann. Codebook: Discovering and exploiting relationships in software repositories. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 125–134, New York, NY, USA, 2010. ACM.

[5] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. Mining email social networks. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 137–143, New York, NY, USA, 2006. ACM.

[6] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: Examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 4–14, New York, NY, USA, 2011. ACM.

[7] G. Bortis and A. v. d. Hoek. Porchlight: A tag-based approach to bug triaging. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 342–351, Piscataway, NJ, USA, 2013. IEEE Press.

[8] M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley. Identification of coordination requirements: Implications for the design of collaboration and awareness tools. In *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*, CSCW '06, pages 353–362, New York, NY, USA, 2006. ACM.

[9] C. S. Corley, E. A. Kammer, and N. A. Kraft. Modeling the ownership of source code topics. In D. Beyer, A. van Deursen, and M. W. Godfrey, editors, *ICPC*, pages 173–182, 2012.

[10] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE*, 41(6):391–407, 1990.

[11] C. Del Rosso. Comprehend and analyze knowledge networks to improve software evolution. *J. Softw. Maint. Evol.*, 21(3):189–215, May 2009.

[12] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 23–32, 2003.

[13] D. German. An empirical study of fine-grained software modifications. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 316–325, 2004.

[14] D. M. German. A study of the contributors of postgresql. In *Proceedings of the 2006 ACM International Workshop on Mining Software Repositories*, MSR '06, pages 163–164, New York, NY, USA, 2006.

[15] 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*, ESEC/FSE '09, pages 111–120, New York, NY, USA, 2009. ACM.

[16] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Hammad. Assigning change requests to software developers. *Journal of Software: Evolution and Process*, 24(1):3–33, 2012.

[17] 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*, pages 451–460, 2012.

[18] D. Ma, D. Schuler, T. Zimmermann, and J. Sillito. Expert recommendation with usage expertise. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 535–538, 2009.

[19] 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*, pages 131–140, 2009.

[20] 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*, CSCW '00, pages 231–240, New York, NY, USA, 2000. ACM.

[21] S. Minto and G. Murphy. Recommending emergent teams. In *Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth International Workshop on*, pages 5–5, 2007.

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

[23] L. Moonen. Lightweight impact analysis using island grammars. In *In Proceedings of the 10th International Workshop on Program Comprehension (IWPC 2002). IEEE Computer*, pages 219–228. Society Press, 2002.

[24] D. Poshyvanyk and A. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on*, pages 37–48, 2007.

[25] F. Rahman and P. Devanbu. Ownership, experience and defects: A fine-grained study of authorship. In *Proceedings of the 33rd International Conference on*

Software Engineering, ICSE '11, pages 491–500, New York, NY, USA, 2011. ACM.

[26] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani. Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation. In *MSR*, pages 2–11, 2013.

[27] X. Song, B. L. Tseng, C. yung Lin, and M. ting Sun. Expertisenet: Relational and evolutionary expert modeling. In *in User Modeling, 2005*, pages 99–108, 2005.

[28] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Fuzzy set and cache-based approach for bug triaging. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 365–375, New York, NY, USA, 2011. ACM.

[29] P. Weissgerber, M. Pohl, and M. Burch. Visual data mining in software archives to detect how developers work together. In *Mining Software Repositories, 2007. ICSE Workshops MSR '07. Fourth International Workshop on*, pages 9–9, 2007.

[30] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: Recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 15–25, New York, NY, USA, 2011. ACM.

[31] X. Xia, D. Lo, X. Wang, and B. Zhou. Accurate developer recommendation for bug resolution. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 72–81, 2013.