AUTOMATED DEVELOPER RECOMMENDATIONS FOR INCOMING SOFTWARE CHANGE REQUESTS

A Thesis by

Md Kamal Hossen

Bachelor of Science, Bangladesh University of Engineering and Technology, 2009

Submitted to the Department of Electrical Engineering and Computer Science
and the faculty of the Graduate School of
Wichita State University
in partial fulfillment of
the requirements for the degree of
Master of Science

May 2014

AUTOMATED DEVELOPER RECOMMENDATIONS FOR INCOMING SOFTWARE CHANGE REUESTS

The following faculty members have examined the final copy of this thesis for form and content, and recommend that it be accepted in partial fulfillment of the requirement for the degree of Master of Science, with a major in Computer Science.

____
Huzefa Kagdi, Committee Chair

____
Yi Song, Committee Member

____
Jibo He, Committee Member

ACKNOWLEDGMENTS

First of all I would like to express my gratitude to my supervisor and thesis committee chair, Dr. Huzefa Kagdi, whose expertise, understanding, and patience, added considerably to my graduate experience. His vast knowledge and research skill in the area of software maintenance and evolution, and his expertise in research publications are really admirable. His motivational sentences like "*Can you finish it by yesterday*" or "*Why not try this*" reminded me that there is always room to improve and helped me to reach the optimal position. Without his supervision and constant help this thesis would not be possible. I really feel lucky to get a friendly adviser like him.

I would also like to thank other thesis committee members who put their valuable time to review my thesis and make it a successful one. My sincere gratitude goes to the faculty members that I came across during my entire graduate program at Wichita State University.

Last but not least, the extraordinary support and prayers I received from my parents, family, and friends have been the most important part of my journey throughout my graduate studies. I would like to extend my special gratitude to them for supporting me in every step of the way.

ABSTRACT

Software change requests, such as bug fixes and new features, are an integral part of software evolution and maintenance. It is not uncommon in open source projects to receive numerous change requests daily, which need to be triaged. Therein, automatically assigning the most appropriate developer(s) to resolve an incoming change request is an important task. The thesis proposes two approaches to address this task. The first approach, namely *iA*, 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.

The second approach, namely, *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 in the literature previously.

An empirical study to evaluate the effectiveness of the approaches on open source systems, *ArgoUML*, *JabRef*, *jEdit*, and *MuCommander*, is reported. The *iA* approach is found to provide recommendation accuracies that are equivalent or better than the two compared approaches. Results also show that *iMacPro* could provide recall gains from 30% to 180% over its subjected competitor with statistical significance.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

ANOVA	Analysis of Variance
SVN	Subversion (Version Control System)

LIST OF SYMBOLS

$\uplus$	Multiset sum
$\cup$	Union
$\in$	Element of
$\emptyset$	Empty set
$\Rightarrow$	Implication
$\mathcal{S}$	Minimum support

CHAPTER 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 daily [1] and effective supporting of those changes is essential to provide a sustainable high quality evolution of large-scale software systems.

The task of automatically assigning issue 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 [1-5]. 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.

We propose two approaches for assigning incoming change requests to appropriate developers. The first approach, namely *iA*, is a novel approach to assign incoming change requests to developer that does not require mining of either a bug or a commit repository. Central to the *iA* 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, or method).

Figure 1 shows that the author *Slava Pestov* is found in the second line of the header comment of the file *Mode.java*. Authors *mike dillon* and *dalenson* are also found in the header comments (they are underlined in red). The premise of our *iA* approach 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 [6] 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.

Our second approach, namely *iMacPro,* uses the same IR based concept location technique as the *iA* approach to find the relevant source code units to a given change request. The relevant source code units are the 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 iMacPro 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, the *iMacPro* approach favors maintainers of the 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 access to the source code and their change history (i.e., commits); however, it does not require any training from the previously resolved bug reports.

To evaluate the accuracy of our two proposed techniques, we conducted an empirical study on open source systems/repositories: *ArgoUML, JabRef, MuCommander* and *jEdit*. Recall values for *iMacPro,* and precision and recall values for *iA*, of the developer recommendations on a number of bug reports sampled from these systems are presented. That is, how effective our two approaches are at recommending the actual developers who ended up fixing these bugs. We compared *iA* approach with two other approaches that require mining of software repositories. The results show that the *iA* approach performs as well as, or better than, the two other competitive approaches in terms of the recommendation accuracies.

Also, our *iMacPro* approach is empirically compared with *iA* approach. Results show that *iMacPro* outperformed *iA*. 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.

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

1. A novel developer recommendation approach *iA* 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.

2. A comparative study of *iA* 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 heavyweight mining approaches.

3. To the best of our knowledge, our *iMacPro* approach is the first to integrate the change proneness, authors, and maintainers of the relevant source code.
4. We performed a comparative study of *iMacPro* approach with our *iA* approach.
5. An empirical assessment of the contributions of the maintainer information toward the overall accuracy of *iMacPro.*

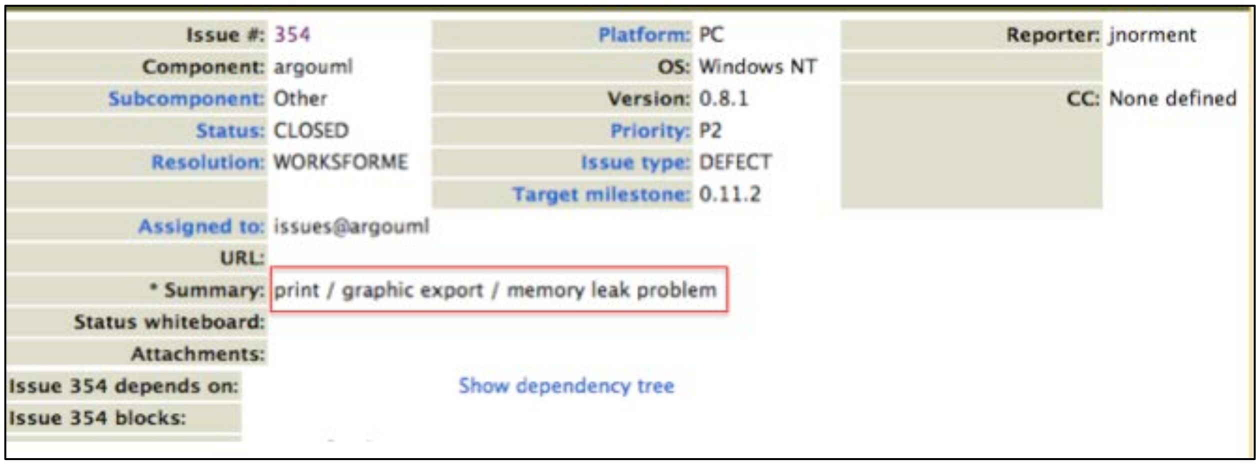
The rest of the thesis is organized as follows: Background and related work is discussed in CHAPTER 2. Our *iA* and *iMacPro* approaches are discussed in CHAPTER 3. The empirical study on open source projects and the results are presented in CHAPTER 4. Threats to validity are discussed in CHAPTER 5. Finally, our conclusions and future work are stated in CHAPTER 6.

CHAPTER 2

BACKGROUND & RELATED WORK

2.1 Background

Large-scale software systems are not stagnant. They are always evolving. Nowadays, every software system has some kind of an issue or bug tracking system where software users can report problems with the software or new features they wish to be included in the software. These reports from issue tracking system are generally called change requests in the software maintenance and evolution terminology. They form a starting point of the software evolution process. Figure 1 shows a typical change request that was reported in the open source system *ArgoUML* with a summary of the issue highlighted in the read rectangle.



Issue #:	354	Platform:	PC	Reporter:	jnorment
Component:	argouml	OS:	Windows NT		
Subcomponent:	Other	Version:	0.8.1	CC:	None defined
Status:	CLOSED	Priority:	P2		
Resolution:	WORKSFORME	Issue type:	DEFECT		
		Target milestone:	0.11.2		
Assigned to:	issues@argouml				
URL:					
* Summary:	print / graphic export / memory leak problem				
Status whiteboard:					
Attachments:					
Issue 354 depends on:		Show dependency tree			
Issue 354 blocks:					

Figure 1 Issue Report from ArgoUML system.

After receiving and validating a change request, software developers modify the software system to address it. A new release of the software system is formed after a set of these change requests is resolved with software changes. An important task in this issue triaging process is assigning the developer(s) with the right expertise to resolve the incoming change request. This task is often referred to as *Developer Recommendation* [1, 3].

Assigning change requests to the developers with the right implementation expertise typically requires project or even organization wide knowledge and the balancing of many factors. Software systems have grown in size and complexity over the years. It is not uncommon for a large-scale open source software system developed collaboratively with hundreds of contributors (often geographically distributed) and years of development history to receive hundreds of change request everyday [1]. One straightforward approach is to email the project team/developers or to use any common communication media to discuss and rely on them for suggestions or advice on who has helpful knowledge about a certain part of source code, a bug, or a feature. But this tedious manual approach is reactive, slow, and may not necessarily yield an effective or efficient answer. Clearly, an automatic approach of developer recommendation is desirable in the issue triaging process. Automated developer recommendation is a very well researched problem in the software maintenance and evolution research community. In section 2.1, we present related work associated with the topic of developer recommendation.

2.1 Related Work

In McDonald and Ackerman [7] 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 [8] developed a tool called Emergent Expertise Locator (EEL), which is based on the framework of Cataldo et al. [9] 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) [10]. 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 [11] 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. [4] 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 recommended 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 [12]. Another approach to facilitate bug triaging uses a graph model based on Markov chains, which captures the bug reassignment history [13]. Matter et al. [14] 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 [15]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. [16] 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 [17]. His findings indicated that in the last years of PostgreSQL only two persons were responsible for most of the source code. Bird et al. [18] analyzed the communication and co-ordination activities of the participants by mining email archives. Del Rosso [19] built a social network of knowledge-intensive software developers based on collaborations and interaction. Ma et al. [20] proposed a technique that uses implementation expertise (i.e., developers usage of API methods) to identify developers. Weissgerber et al. [21] 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 [22] provided a visualization to show which developers tend to modify certain files by studying the modification records (MRs) of CVS logs. Fischer et al. [23] analyzed and related bug report data for tracking features in software. Bortis et al. [24] 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. [25] 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. [26] 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. [27]conducted a survey of inter-team coordination needs and presented a flexible *Codebook* framework that can address most of those needs.

CHAPTER 3

PROPOSED APPROACHES TOWARD DEVELOPER RECOMMENDATION: *iA* AND *iMacPro*

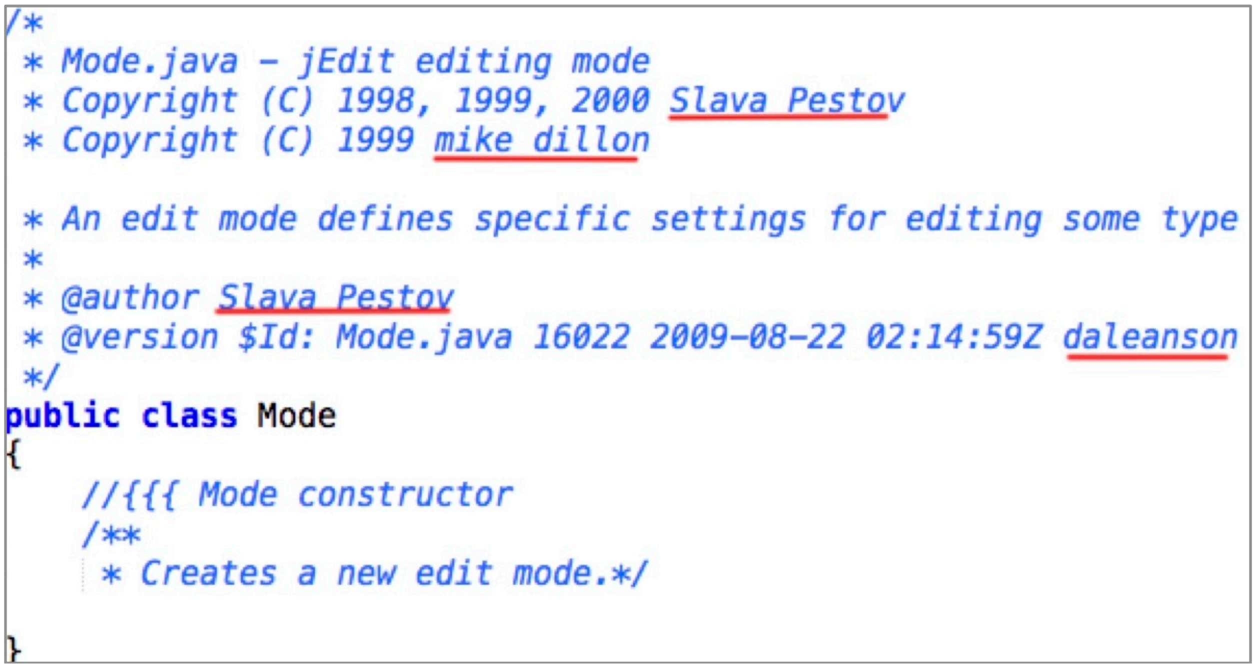
Before we proceed with the finer details of our approach, key terms are defined and discussed next.

3.1 Key Terms and Definitions

Author: Authors of 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 2 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 2. The extraction of authors from source code is discussed in Section 3.5.

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.



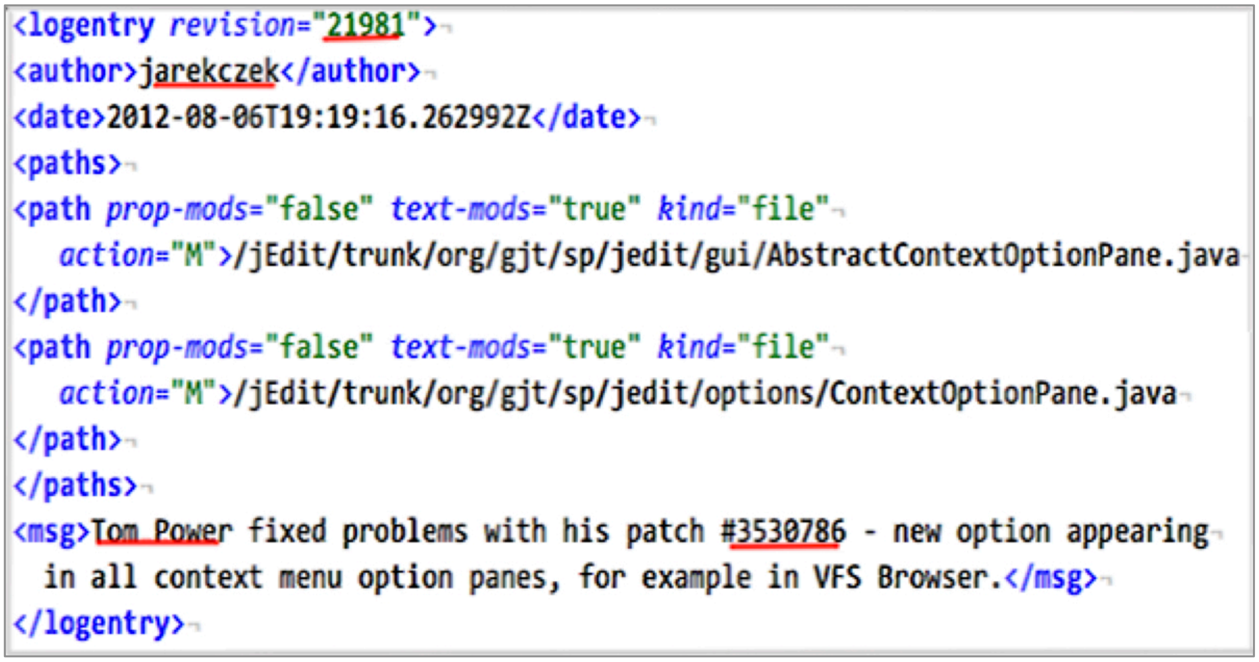
```
/*
 * Mode.java - jEdit editing mode
 * Copyright (C) 1998, 1999, 2000 Slava Pestov
 * Copyright (C) 1999 mike dillon

 * 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 2. 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.

Figure 3 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 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 3.6.



```
<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 3. 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.

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 IFCs. The commit log shown in Figure 3 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 is given by the number of *IFC*s in the commit history that contains it.

$$ICP(e) = |\{\forall_c \in IFC_s | e \in c\}| \qquad (1)$$

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.

3.2 Overview of Developer Recommendation approaches

Here we briefly described our two developer recommendation approaches before going details of them.

3.2.1 The *iA* approach

The *iA* approach of triaging incoming change requests consists of the following two steps:

1. We use Latent Semantic Indexing (LSI) [28] 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. 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).

3.2.2 The *iMacPro* Approach

The *iMacPro* approach to assign incoming change requests to appropriate developers consists of the following steps:

1. First approach of locating relevant source code file is same as the first step of *iA* approach described in chapter 3.2.1
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 3.4 and 3.6).
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.

3.3 Locating Relevant Files with Information Retrieval

In our approaches, in order to locate textually relevant files, we rely on an IR-based concept location techniques [29]. 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 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.

For example change request of interest here is the bug# 4563 of *ArogUML* system, 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

3.4 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 $f_1, f_2, \ldots f_n$ *for* the bug b_i. 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. 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 3.1* 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 show 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. Top five files relevant to Bug#4563 in *ArgoUML*, i.e., it is ranked ahead.

TABLE 2. THE AUTHORS AND MAINTAINERS EXTRACTED FROM EACH OF THE TOP FIVE FILES RELEVANT TO ISSUE# 4563.

Rank	Files	ICP	Ranked Maintainers(*Dm*)	Ranked Authors(*Da*)	Combined Developers(*Dma*)
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, jrob-bins, kataka
3	. . /FigRealization.java	4	mkl, linus, kataka	tfmorris	mkl, tfmorris, linus, kataka
4	. . /CrMultipleRealization.java	4	mkl, linus	mvw, jrobbins	mkl, mvw, linus, jrob-bins
5	. . /FigAbstraction.java	1	mvw	tfmorris, agauthie	mvw, tfmorris, agau-thie

3.5 Extracting Authors from Source Code

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 in our approach; however, this element can be easily replaced by any lightweight source code analysis methods, including regular expressions or island grammar [30].

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 Michiel Vander Wulp (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 2 shows that the author *Slava Pestov* is found in the first line of the header comment of the file Mode.java. Thus, the author *Slava Pestov* 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*].

3.6 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 3.1). 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.

3.7 *iA* Recommendation

We now describe how source files author information are used to recommend a ranked list of developers for incoming change request. 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* relevant files given by LSI step. 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. 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.

Table 3 shows the ranked list of developers produced after the application of the frequency based ranking mechanism of *iA* approach on Top 5 relevant files of Bug#4563 in Table 1. The developer tfmorris and jrobbins appear in two files (see Table 2 for the specific files). But tfmorris end up at number one positon because tfmorris found in the file, which has ranked higher in Table 1 than the file in which jrobbins found. The developers mvw,linus,bobtarling and agauthie are tied because they all appear in one file. These developers are sorted based on the file rank in which they first occur. So, the final list of recommended developers for Bug#4563 given by *iA* approach is [tfmorris, jrobbins, mvw, linus, bobtarling, agauthie].

TABLE 3. DEVELOPER FREQUENCY COUNT

Developer ID	File Frequency
tfmorris	2
jrobins	2
mvw	1
linus	1
bobtarling	1
agauthie	1

3.8 *iMacPro* Recommendation

We now describe the details of *iMacPro* - 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 3.5, there is a one-to-many relationship between the source code file and authors. That is, each file *fi* 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

TABLE 4. DEVELOPERS TAKEN FROM INDIVIDUAL FILES IN POSITIONS 1 TO 5 FROM TABLE 2. THE STRIKEOUT DEVEL OPERS ARE DISCARDED AND OTHERS ARE RETAINED.

Position	Combined Developers (Df)
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
Df	mvw, mkl, bobtarling, linus, tfmorris, agauthie, jrobbins, kataka
$Df@k$=5	mvw, mkl, bobtarling, linus, tfmorris

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:

$$D_a = \begin{pmatrix} f_1 & D_{af_i} \\ f_2 & D_{af_2} \\ \ddot{} & \ddot{} \\ f_n & D_{af_n} \end{pmatrix} D_{af_i} = [a_1 \;\; a_2 \ldots a_l] \tag{2}$$

In Equation (2), D_{afi} represents the ranked list of authors, with no duplication, for the file *fi*. 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 fi 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 maintainers. 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:

$$D_a = \begin{pmatrix} f_1 & D_{mf_i} \\ f_2 & D_{mf_2} \\ \ddot{} & \ddot{} \\ f_n & D_{mf_n} \end{pmatrix} D_{mf_i} = [m_1 \;\; m_2 \ldots m_o] \tag{3}$$

In Equation (3), D_{mfi} 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 *fi*, i.e., D_{ma}, ranked lists of maintainers (D_m) and authors (D_a) are assembled.

$$D_{ma} = D_m \uplus D_a \tag{4}$$

$$= \begin{pmatrix} f_1\left[d_1\, d_2 \,\dots\, d_{\max(l,o)}\right] \\ f_2\left[d_1\, d_2 \,\dots\, d_{\max(l,o)}\right] \\ \vdots \\ f_n\left[d_1\, d_2 \,\dots\, d_{\max(l,o)}\right] \end{pmatrix} \tag{5}$$

We employ a round-robin merging algorithm. For each file *fi*, 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 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. 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.

$$D_{f=}d|\uplus_{i=1}^{n}\forall_{d}\in D_{mafi}\in\ D_{ma} \tag{6}$$

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 4 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 *Df @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 compare to *iA* approach three recommendation.

CHAPTER 4

EVALUATION AND RESULTS

Now we describe case studies two evaluate our two proposed developer recommendation approach.

4.1 *iA* Approach Evaluation and Results

The purpose of this empirical study was to investigate how well our *iA* approach recommends expert developers to assist with incoming change requests. We also compared our *iA* approach with two previously published approaches. The first approach is based on the mining of a bug report history by Anvik et al. [1]), denoted here as machine learning - *ML*. The second is based on mining of source control repositories, *i.e.*, commit logs, by Kagdi et al.[3] denoted here as *xFinder*. Therefore, we addressed the following research questions (RQ) in our case study:

- **RQ1:** How does the accuracy of the *iA* approach compare to its two competitors that are based on software repository mining, namely ML and xFinder?
- **RQ2:** Is there any impact on the results of *iA* when filtering of IR-based results with dynamic-analysis information is included, *i.e.*, an additional analysis cost is incurred?

The rationale behind **RQ** is two-fold: 1) To assess whether our *iA* can identify correct developers to handle change requests in open source systems, and 2) how well the accuracy of the *iA* approach compares to the *ML* and *xFinder* approaches. The purpose of **RQ2** is to assess if incorporating an additional software analysis technique to the first step of the *iA* approach improves its accuracy results.

We used a dynamic analysis technique because it was found to improve the accuracy of IR-based feature location and impact analysis approaches [31, 32]. 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 *iA* without the dynamic filtering.

Next, we provide background information on the two competitive approaches *ML* and *xFinder* 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 *tfidf* measure on the remaining words. Neither stemming nor attributes selection methods were applied [1].

In order to compare our *iA* 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, *tfidf* 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 (*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 [33], software categorization [34], and developers recommendation [1, 35].

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 *iA* approach (see Section 3.2.1).
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 [36]. 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 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*.

4.1.1 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 in Table 5.

Table 5. SUBJECT SOFTWARE SYSTEMS USED IN THE *iA* Approach 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
muComma	0.8.5	76,649	1,069	8,187	4,262

4.1.2 Building The Benchmark

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 6. Also, the minimum, mean, maximum number of developers for the considered change requests is 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 committer was different from the actual developer who contributed changes. The actual developer was mentioned in the commit comments, included in our gold set.

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.

TABLE 6. SUMMARY OF THE BENCHMARKS FOR *IA* APPROACH

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

4.1.3 Collecting and Using Execution Information

The idea of integrating IR with dynamic analysis was previously defined in the context of feature location [37]; 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 runtime 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* [31, 32]. We denote a version of our approach that uses execution information as *iA*$_F$. 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].

4.1.4 Metrics and Statistical Analyses

We evaluated the accuracy of each one of the approaches, for all the reports in our testing set, using the 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 [38]. 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):

- **H0-1**: There is no statistically significant difference between the **precision/recall** of *ML* and *iA*.

- **H0-2**: There is no statistically significant difference between the **precision/recall** of *xFinder* and *iA*.
- **H0-3**: There is no statistically significant difference between the **precision/recall** of *xFinder*$_F$ and *iA*.
- **H0-4**: There is no statistically significant difference between the **precision/recall** of *ML* and *iA*$_F$.
- **H0-1**: There is no statistically significant difference between the **precision/recall** of *ML* and *iA*.

- **H0-2**: There is no statistically significant difference between the **precision/recall** of *xFinder* and *iA*.
- **H0-3**: There is no statistically significant difference between the **precision/recall** of *xFinder*$_F$ and *iA*.
- **H0-4**: There is no statistically significant difference between the **precision/recall** of *ML* and *iA*$_{F.}$

- **H**$_{0\text{-}5}$: There is no statistically significant difference between the **precision/recall** of *xFinder* and *iA*$_F$.
- **H**$_{0\text{-}6}$: There is no statistically significant difference between the **precision/recall** of *xFinder*$_F$ and *iA*$_F$.
- **H**$_{0\text{-}7}$: There is no statistically significant difference between the **precision/recall** of *iA*$_F$ and *iA*.

The hypotheses from H0-1 up to H0-6 were used to answer RQ1, and H0-7 was used to answer RQ2.

4.1.5 *iA* Approach Results

Figure 4 depicts the average precision and recalls for the three systems3. For top-1 recommendations, we found that *iA* 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*; *iA* had the best accuracy for MuCommander from top-2 to top-10.

For top-1 recommendations in *ArgoUML* (Figure 4.a), the *iA* provided the highest accuracy. However, we did not found statistical significant difference between the accuracies of the *iA* and the other techniques. One possible explanation is that the acceptable precision values for top-1 recommendations are either zero or one, and the *iA* 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 *iA* and *ML* is 19%, the distribution of zeros and ones in both approaches is very similar.

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

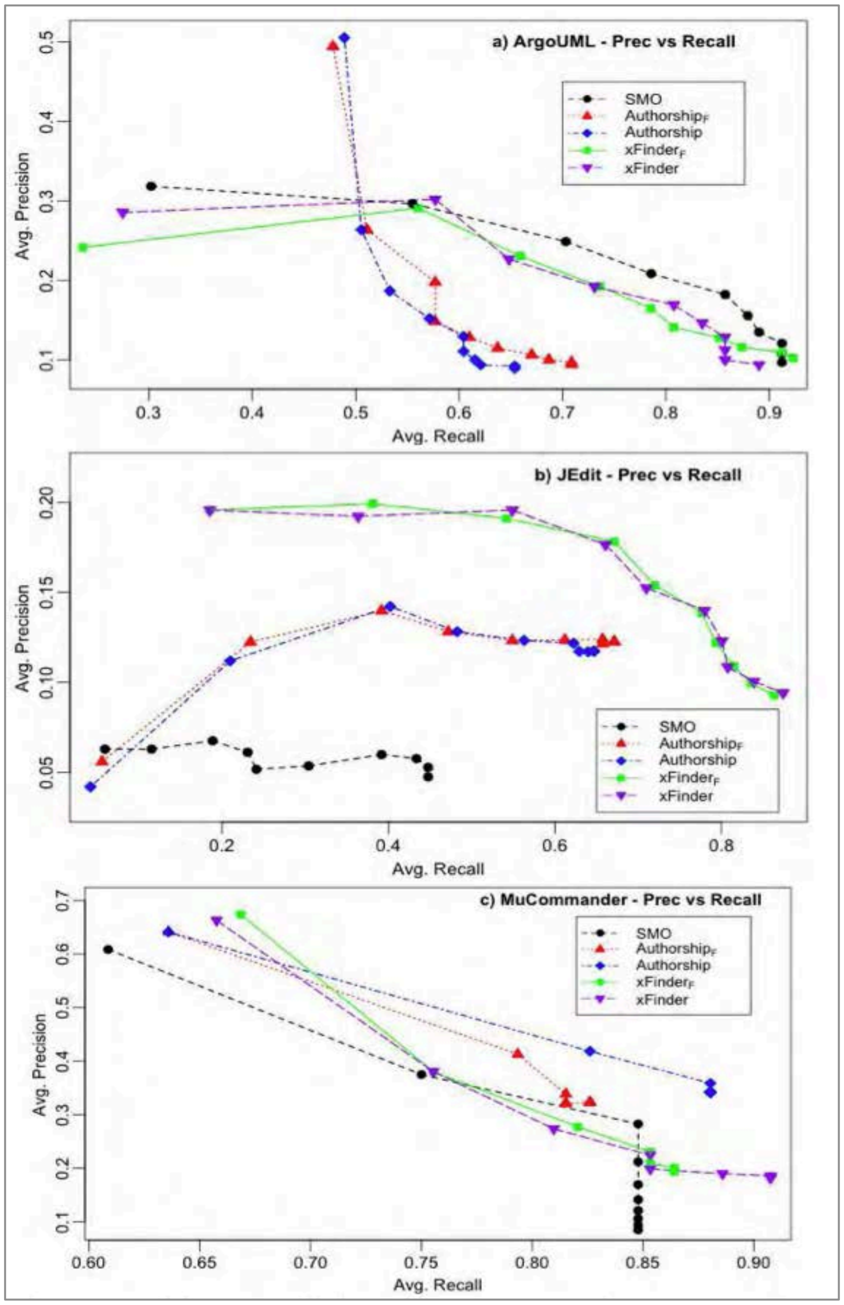


Figure 4. Precision vs. recall charts for *ArgoUML*, *JEdit*, *MuCommander*. These results are for four approaches (*ML* – SMO, *xFinder*, *xFinder$_F$*, *iA*, and *iA$_F$*). Each curve has a point for each recommendation from top-1 to10.

Although the difference between the precision of the *iA* 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 *iA* 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. *iA* ranged from 0.8% to 4% with mean 2.5%, and for *ML* vs. *iA* ranged from 0.3% to 6% with mean 3.4%; the difference in recall from top-2 to top-10 for *xFinder* vs. *iA* ranged from 4.9% to 21.4% with mean 15.8%, and for *ML* vs. *iA* ranged from 4.4% to 24.7% with mean 19.1%. The reason behind this sharp decline in the *iA* 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.

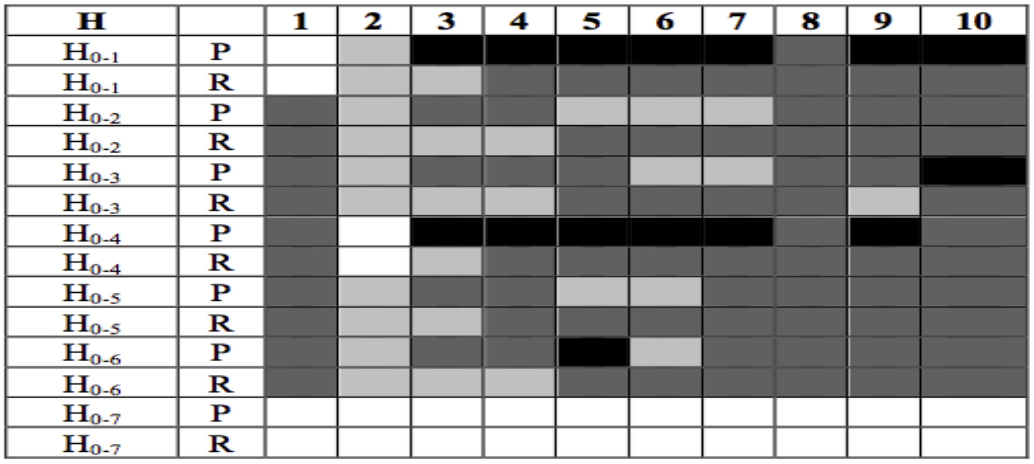
For *JEdit* (Figure 4.b), *xFinder* had a higher accuracy than *ML* and *iA* from top-1 to top-10 recommendations with a significant difference (except for top-5 and top-6 precision). *iA* exhibited higher accuracy than *ML* from top-2 to top-10 recommendations with a difference from top-3 to top-10; the difference in precision from top-2 to top-10 for *iA* 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 4.c), the *iA* 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 *iA* vs. *ML* ranged from 3.8% to 23.8% with mean 15.8%.

The *iA* 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 (Figure 5).

Therefore, for **RQ1** we concluded that the **precision of the *iA* outperformed *ML* on *JEdit* and *MuCommander* datasets.**

iA 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 **RQ1**, we concluded that the **precision of the *iA* outperformed *xFinder* on *MuCommander*.**



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	□	□	□	□	□	□	□	□	□	□

Figure 5. HEAT-MAP summarizing results for testing hypothesis across all the system. 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.

For **RQ2,** we did not find a conclusive support for a significant difference in the accuracies of *iA and iA$_F$*. We could not reject **H0-7** in any of the systems. Therefore, we concluded that ***iA performs as well as iA$_F$* in terms of accuracy**. *These results suggest that the additional overhead of dynamic analysis in the iA and xFinder was not justified, as there was no statistically significant accuracy gain.*

Now, we provide representative bugs from the three systems detailing where *iA* outperformed the other approaches. For example, *iA* 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 *iA* 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, *iA* 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. *iA* and *ML* obtained 100% precision with top-1 recommendation for the bug report# 277 in *MuCommander*, however, the *xFinderF* achieved its highest value of precision of 50% using a recommendation with five developers.

4.2 *iMacPro* Evaluation and Results

The main purpose of this case study was to investigate how well our *iMacPro* approach recommends expert developers to assist with incoming change requests. Moreover, we compare *iMacPro* with *iA* approach. Similar to *iA*, *iMacPro* uses LSI to identify relevant source files and then uses the author information in those files for recommending developers. However, it uses 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 (RQs) in our case study:

- **RQ1:** What is the accuracy of *iMacPro* when assessed on open-source systems?
- **RQ2:** How does the accuracy of *iMacPro* compare to *iA*.
- **RQ3:** 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?

4.2.1 *iAcPro* Approach

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 approaches 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?

4.2.2 Subjected 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 [5]. 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 7.

TABLE 7. SUBJECT SOFTWARE SYSTEM USED IN *iMacPro* 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

4.2.3 Building Benchmark

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

4.2.4 Metric and Statistical Analysis

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 [2, 5, 11]. 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:

$$recall@k = \frac{1}{b}\frac{\sum_{i=1}^{b}|RD(b_i) \cap AD(b_i)|}{|AD(b_i)|} \qquad (7)$$

where *RD(bi)* and *AD(bi)* are the recommended developer by the approach and the actual developer who resolved the issue for the bug *bi*. 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 be- cause 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 [5]. 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 be- tween 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*.

4.2.5 *iMacPro* Approach Results

For each change request of each subject system in the benchmark, we parameterized 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 8shows 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.

TABLE 8. 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

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:

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

The iMacPro gain over *iA* % column in TABLE 8 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 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:

$$gain@k_{iMacPro-iAcPro} = \frac{recall@k_{iMacPro} - recall@k_{iAcPro}}{recall@k_{iAcPro}} \times 100 \qquad (9)$$

The iMacPro gain over *iAcPro* % column in TABLE 8 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 H1, 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 9 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 ≤0.05, so we can reject the null hypothesis H1.

In the case of ArgoUML, the p-values are ≤0.05 for k = 5 and k = 10, so we can reject the null hypothesis H1. Note that in these case, the reported gains were positive in TABLE 8. 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 is sues 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*.

TABLE 9. 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

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 9 shows the p-values for k=1,k=5,andk=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 1st 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 1st position, whereas, *iA* and *iAcPro* found him in the the 2nd position. In *ArgoUML*, iA and *iAcPro* recommended the correct developer (tfmorris) for bug#4720 in the 5ith and 3rd positions respectively, whereas, *iMacPro* found it in the 1st position.

CHAPTER 5

THREATS TO VALIDITY

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

5.1 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 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 stake- holders, 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.

5.2 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. [39] identified the missing traceability between bug reports and commits in Apache. Wu et al. [40] 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 maintainer: 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.

5.3 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 and other noisy information in bug/issue databases [18, 27]. 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.

5.4 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. 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.

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.

CHAPTER 6

CONCLUSION AND FUTURE WORK

We presented the *iA* and *iMacPro* approach to recommend developers who are most likely to implement incoming change requests. To the best of our knowledge, *iA* 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. *iA* approach is perhaps simple and lightweight. Nonetheless, our empirical evaluation shows that it can be quite effective and competitive with the other history-based approaches.

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.

REFERENCES

REFERENCES

[1] J. Anvik, L. Hiew, and G. C. Murphy, "Who Should Fix This Bug?," in *proceedings of 28th ACM International Conference on Software Engineering, 2006*, pp. 361-370.

[2] X. Xia, D. Lo, X. Wang, and B. Zhou, "Accurate developer recommendation for bug resolution," in *Proceedings of 20th Working Conference on Reverse Engineering (WCRE), 2013*, pp. 72-81.

[3] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Hammad, "Assigning change requests to software developers," *Journal of Software: Evolution and Process,* vol. 24, pp. 3-33, 2012.

[4] 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 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering,2011*, pp. 365-375.

[5] 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 *proceedings of 28th IEEE International Conference on Software Maintenance (ICSM), 2012*, pp. 451-460.

[6] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process,* vol. 25, pp. 53-95, 2013.

[7] D. W. McDonald and M. S. Ackerman, "Expertise Recommender: A Flexible Recommendation System and Architecture," in *proceedings of 2000 ACM Conference on Computer Supported Cooperative Work*, pp. 231-240.

[8] S. Minto and G. C. Murphy, "Recommending Emergent Teams," in *proceedings of fourth International Workshop on Mining Software Repositories, 2007. ICSE Workshops MSR '07.*, pp. 5-5.

[9] 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 2006 20th Anniversary Conference on Computer Supported Cooperative Work*, pp. 353-362.

[10] A. Mockus and J. D. Herbsleb, "Expertise Browser: A Quantitative Approach to Identifying Expertise," in *proceedings of 24th International Conference on Software Engineering, 2002*, pp. 503-512.

[11] J. Anvik and G. C. Murphy, "Determining Implementation Expertise from Bug Reports," in *proceedings of fourth International Workshop on Mining Software Repositories (MSR), 2007 ICSE Workshops MSR '07*, pp. 2-2.

[12] X. Song, B. L. Tseng, C. yung Lin, and M. ting Sun, "Expertisenet: Relational and evolutionary expert modeling," in *proceedings of User Modeling, 2005*, pp. 99-108.

[13] 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,2009*, pp. 111-120.

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

[15] F. Rahman and P. Devanbu, "Ownership, Experience and Defects: A Fine-grained Study of Authorship," in *proceedings of 33rd International Conference on Software Engineering,2011*, pp. 491-500.

[16] 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 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering,2011*, pp. 4-14.

[17] D. M. German, "A Study of the Contributors of PostgreSQL," in *proceedings of 2006 ACM International Workshop on Mining Software Repositories*, pp. 163-164.

[18] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining Email Social Networks," in *proceedings of 2006 International Workshop on Mining Software Repositories*, pp. 137-143.

[19] C. Del Rosso, "Comprehend and Analyze Knowledge Networks to Improve Software Evolution," *J. Softw. Maint. Evol.*, vol. 21, pp. 189-215, 2009.

[20] D. Ma, D. Schuler, T. Zimmermann, and J. Sillito, "Expert recommendation with usage expertise," in *proceedings of IEEE International Conference on Software Maintenance,ICSM 2009.*, pp. 535-538.

[21] P. Weissgerber, M. Pohl, and M. Burch, "Visual Data Mining in Software Archives to Detect How Developers Work Together," in *proceedings of fourth International Workshop on Mining Software Repositories, 2007. ICSE Workshops MSR '07.*, pp. 9-9.

[22] D. M. German, "An empirical study of fine-grained software modifications," in *proceedings of 20th IEEE International Conference on Software Maintenance, 2004.*, pp. 316-325.

[23] M. Fischer, M. Pinzger, and H. Gall, "Populating a Release History Database from version control and bug tracking systems," in *proceedings of International Conference on Software Maintenance, 2003. ICSM 2003.*, pp. 23-32.

[24] G. Bortis and A. v. d. Hoek, "PorchLight: A Tag-based Approach to Bug Triaging," in *proceedings of 2013 International Conference on Software Engineering*, pp. 342-351.

[25] 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 *proceedings of 10th IEEE Working Conference on Mining Software Repositories (MSR), 2013*, pp. 2-11.

[26] C. S. Corley, E. A. Kammer, and N. A. Kraft, "Modeling the ownership of source code topics," in *proceedings of IEEE 20th International Conference on Program Comprehension (ICPC), 2012*, pp. 173-182.

[27] A. Begel, Y. P. Khoo, and T. Zimmermann, "Codebook: Discovering and Exploiting Relationships in Software Repositories," in *proceedings of 32Nd ACM/IEEE International Conference on Software Engineering, 2010 - Volume 1*, pp. 125-134.

[28] 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,* vol. 41, pp. 391-407, 1990.

[29] D. Poshyvanyk and A. Marcus, "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code," in *proceedings of 15th IEEE International Conference on Program Comprehension, 2007. ICPC '07.*, pp. 37-48.

[30] L. Moonen, "Lightweight Impact Analysis using Island Grammars," in *proceedings of 10th International Workshop on Program Comprehension (IWPC 2002). IEEE Computer*, pp. 219-228.

[31] D. Poshyvanyk, Y. gaël Guéhéneuc, and A. Marcus, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *IEEE Trans. Software Eng,* pp. 432-432, 2007.

[32] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk, "Integrated Impact Analysis for Managing Software Changes," in *proceedings of 2012 International Conference on Software Engineering*, pp. 430-440.

[33] E. Leopold and J. Kindermann, "Text Categorization with Support Vector Machines. How to Represent Texts in Input Space?," *Mach. Learn.,* vol. 46, pp. 423-444, 2002.

[34] C. McMillan, M. Linares-Vasquez, D. Poshyvanyk, and M. Grechanik, "Categorizing software applications for maintenance," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pp. 343-352.

[35] J. Anvik and G. C. Murphy, "Reducing the Effort of Bug Report Triage: Recommenders for Development-oriented Decisions," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, pp. 10:1-10:35, 2011.

[36] H. Kagdi, M. Hammad, and J. I. Maletic, "Who can help me with this source code change?," in *proceedings of IEEE International Conference on Software Maintenance (ICSM 2008)*, pp. 157-166.

[37] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace," in *proceedings of Automated Software Engineering (ASE 2007)*, pp. 234243-234243.

[38] T. Hettmansperger and J. McKean, "Statistical inference based on ranks," *Psychometrika*, vol. 43, pp. 69-79, 1978.

[39] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The Missing Links: Bugs and Bug-fix Commits," in *proceedings of Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010*, pp. 97-106.

[40] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "ReLink: Recovering Links Between Bugs and Changes," in *Proceedings of 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, 2011*, pp. 15-25.