Using Developer-Interaction Trails to Triage Change Requests

Motahareh Bahrami Zanjani, Huzefa Kagdi
Department of Electrical Engineering and Computer Science
Wichita State University
Wichita, Kansas 6760, USA
Email: {mxbahramizanjani, huzefa.kagdi}@wichita.edu

Christian Bird
Microsoft Research
Redmond, WA, USA
Email: cbird@microsoft.com

***Abstract*— The paper presents an approach, namely *iHDev*, to recommend developers who are most likely to implement incoming change requests. The basic premise of *iHDev* is that the developers who interacted with the source code relevant to a given change request are most likely to best assist with its resolution. A machine-learning technique is first used to locate source code entities relevant to the textual description of a given change request. *iHDev* then mines interaction trails (i.e., *Mylyn* sessions) associated with these source code entities to recommend a ranked list of developers. *iHDev* integrates the interaction trails in a unique way to perform its task, which was not investigated previously.**

An empirical study on open source systems *Mylyn* and *Eclipse Project* was conducted to assess the effectiveness of *iHDev*. A number of change requests were used in the evaluated benchmark. Recall for top one to five recommended developers and Mean Reciprocal Rank (MRR) values are reported. Furthermore, a comparative study with two previous approaches that use commit histories and/or the source code authorship information for developer recommendation was performed. Results show that *iHDev* could provide a recall gain of up to 127.27% with equivalent or improved MRR values by up to 112.5%.

I. Introduction

Software change requests and their resolution are an integral part of software maintenance and evolution. It is not uncommon in open source projects to receive tens of change requests daily that need to be promptly resolved [1]. Issue triage is a crucial activity in addressing change requests in an effective manner (e.g., within time, priority, and quality factors). The task of automatically assigning issues or change requests to the developer(s) who are most likely to resolve them has been studied under the umbrella of bug or issue triaging. A number of approaches to address this task have been presented in the literature [1]–[5]. The fundamental idea underlying most triage approaches is to identify the expertise and interests of developers, infer the concern or component that must be addressed for a task, and then match developers to tasks. Approaches typically operate on the information available from software repositories (e.g., models trained from past bugs/issues or source code changes), the source code authorship [3], or their combinations [6] under the rationale that if a developer frequently commits changes to, or fixes bugs in, particular parts of a system, they have knowledge of that area and can competently make changes to it.

We agree with the fundamental concept that historical records of developers' activity yield insight into their knowledge and ability, but we also posit that additional traces of developer activity such as interactions with source code (beyond those leading to commits) when resolving an issue can provide a more comprehensive picture of their expertise. The records of developers' interactions with code in resolving an issue remain largely untapped in solving this problem.

In this paper, we show that exploiting these additional sources of such interactions leads to better task assignment than relying on commit and bug tracking histories. We propose a new approach, namely *iHDev*, for assigning the incoming change requests to appropriate developers. *iHDev* is centered on the developers' interactions with source code entities that were involved in the resolution of previous change requests. Developers may interact with source code entities within an Integrated Development Environment (IDE) that may or may not be eventually committed to the code repository [7], [8]. These interactions (e.g., navigate, view, and modify) could have contributed in locating and/or verifying the entities that were changed due to a change request. Therefore, it suggests that the interacting developers are knowledgeable in those entities. On the face value, it could be conjectured that commits (i.e., changed entities) are a subset of interactions (i.e., viewed and changed entities). Our previous investigation found that a superset or subset relationship does not always hold [7]. Thus, the interaction trails have the potential to offer aspects that may not be embodied in commit histories. Tools such as *Mylyn* capture and store such interaction trails (histories) [9].

iHDev takes the textual description of an incoming change request (e.g., a short bug description) and locates relevant entities (e.g., files) from a source code snapshot. A machine learning technique, K-Nearest Neighbor (KNN) algorithm and cosine similarity, is used in this step. The interaction histories of these entities are mined to forge a ranked list of candidate developers to resolve the change request. *The basic premise of our approach is that the developers who interacted with the relevant source code to a given change request in the past are most likely to best assist with its resolution.* In a nutshell, our approach favors **i**nteraction **H**istories over other types of past information to recommend **Dev**elopers; hence, the name *iHDev*. It neither needs to mine for textually

978-0-7695-5594-2/15 $31.00 © 2015 IEEE
DOI 10.1109/MSR.2015.16

similar past change requests nor source code change (commit) histories. It only needs developer-interaction sessions from the issue repository of a system, which are typically attached to issue/bug reports (e.g., in the *Mylyn* prescribed XML format).

To evaluate the accuracy of our technique, we conducted an empirical study on two open source systems *Mylyn* and *Eclipse Project*. Recall and Mean Reciprocal Rank (MRR) metric values of the developer recommendations on a number of bug reports sampled from this system are presented. That is, how effective our *iHDev* approach is at recommending the actual developer who ended up fixing these bugs. Additionally, our *iHDev* approach is empirically compared with two other approaches that use the commit and/or source code authorship information [3], [6], [2]. The results show that the proposed *iHDev* approach outperformed these baseline competitors. Lowest recall gains of 6.17% and 9.72% were recorded against the two respective approaches. Highest recall gains of 125% and 127.27% were recorded against the two respective approaches. These gains came without incurring any decreased Mean Reciprocal Rank (MRR) values; rather *iHDev* recorded improvements in them. That is, *iHDev* would typically recommend the correct developers at higher ranks than the subjected competitors.

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

1) To the best of our knowledge, our *iHDev* approach is the first to utilize developers' source code interaction histories involved with past change requests.
2) We performed a comparative study with two other approaches that use commit and/or source code authorship information.

The rest of the paper is organized as follows: Our approach is discussed in Section II. The empirical study on *Mylyn* and *Eclipse Project*, and its results are presented in Section III. Threats to validity are listed and analyzed in Section IV. Related work is discussed in Section V. Finally, our conclusions and future work are stated in Section VI.

II. Approach

Our approach *iHDev* to assign an incoming change request to the appropriate developer(s) consists of the following steps:

1) **Locating Relevant Entities to Change Request**: We use the K-Nearest Neighbor (KNN) algorithm to locate relevant units of source code (e.g., files and classes) 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 resolved (e.g., a bug is fixed).
2) **Mining Interaction Histories to Recommend Developers**: The interaction histories of the 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). The interaction histories are extracted from the issue-tracking system.

A. Key Terms and Definition

`Interaction:` Interaction is the activity of programmers in an IDE during a development session (e.g., editing a file or referencing an API documentation).

Tools, such as *Mylyn*, have been developed to model programmers' actions in IDEs [9]. *Mylyn* monitors programmers' activities inside the *Eclipse IDE* and uses the data to create an *Eclipse* user interface focused around a task. The *Mylyn* interaction consists of traces of interaction histories. Each historical record encapsulates a set of interaction events needed to complete a task (e.g., a bug fix). Once a task is defined and activated, the *Mylyn* monitor records all the interaction events (the smallest unit of interaction within an IDE) for the active task. For each interaction, the monitor captures about eleven different types of data attributes. The most important of these is the structure handle attribute, which contains a unique identifier for the target element affected by the interaction. For example, the identifier of a Java class contains the names of the package, the file to which the class belongs to, and the class. Similarly, the identifier of a Java method contains the names of the package, the file and the class the method belongs to, the method name, and the parameter type(s) of the method. Figure 1 shows an example of the *Mylyn* interaction edit event.

`Trace file:` For each active task, *Mylyn* creates an XML trace file, typically named *Mylyn*-context.zip or its derivative. A trace file contains the interaction history of a task. This file is typically attached to the project's issue tracking system (e.g., *Bugzilla* or JIRA). The trace files for the *Mylyn* project are archived in the *Bugzilla* as attachments to a bug report. For example for issue #315184 there is one trace file named *Mylyn*-context.zip[1].

`Attacher:` Each issue/bug could have trace files. A developer submits these trace files to the project's issue tracking system and are attached to the associated issue report. We term this developer as the attacher. This term helps differentiate from the use of committers and developers in the context of source code repositories. There is no explicit information to distinguish between the attacher and the actual developer who performed the interaction session. We assume that the attacher is the developer who performed the attached interaction session. This issue is similar to the distinction between the developer who performed and the committer who committed the changes to a source code repository. For example, *Steffen Pingel* attached the file *Mylyn*-context.zip for issue #315184. Considering the *Mylyn* workflow for interactions, we did not expect nor found any instances where the attacher was not the developer who performed the interaction session.

B. Locating Relevant Entities to Change Request

In our approach, we use techniques from natural language processing and machine learning to locate textually relevant source code files to a given change request for which we need to assign developers. The specific steps are given below:

1) Creating a corpus from software: The source code of a release, in or before which the change request is resolved, is parsed using a developer-defined granularity level (e.g.,

[1]https://bugs.eclipse.org/bugs/show_bug.cgi?id=315184

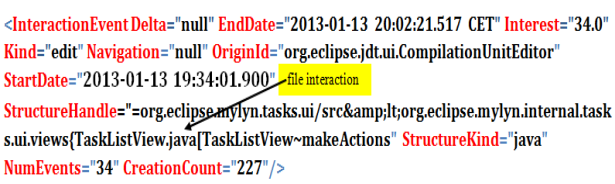

Fig. 1: A snippet of an interaction event recorded by *Mylyn* for bug issue #330695 with trace ID #221752. File TaskListView.java is edited.

file) and documents are extracted. A corpus is created such that each file will have a corresponding document therein. Identifiers, comments, and expressions are extracted from the source code. Each document in the corpus is referred to as doc_{past}.

2) Preprocessing Step: Each document doc_{past} is preprocessed in two parts: removing stop words and stemming.

3) Document-Term Representation: `Document Indexing`: We produce a dictionary from all of the terms in our document and assign a unique integer Id to each term appearing in it.

`Term Weighting`: We use *tfidf* in our approach. The global importance of a term *i* is based on its inverse document frequency (*idf*), calculated as the reciprocal of the number of documents that the term appears in. *idf* is the document frequency of term *i* in the whole document collection. $tf_{i,j}$ is the term frequency of the term *i* in the document *j*. Each document d_j is represented as a vector $d_j = (w_{1,j}, ..., w_{i,j}, ..., w_{n,j})$ where n is the total number of terms in our document collection and $w_{i,j}$ is the weight of the term i in document j.

4) Using Change Request: The textual description of a change request for which we want to find all of the relevant files, and eventually to be assigned developer(s), is referred to as doc_{new}. doc_{new} also goes through the preprocessing step and is represented as a document.

5) K-Nearest-Neighbor: The task of multi-label classification is to predict for each data instance a set of labels that applies to it. Standard classification only assigns one label to each data instance. However, in many settings a data instance can be assigned by more than one label. In our context, each data instance (i.e., a bug report) can be assigned multiple labels (i.e., source code files). ML-KNN is a state-of-the-art algorithm in the multi-label classification literature [10]. We employ the ML-KNN search with a user-defined value of K (e.g., the value of 10 as used in previous work [5]) to search the existing corpus (doc_{past}) based on similarities with the new bug description (doc_{new}). This search finds the top K similar files. We consider these top K files to be a set, i.e., they are eventually relevant to the change request regardless of their similarly levels or ranking. Cosine is then used to measure the similarity of the two document vectors.

$$f_{sim}\left(doc_{new}, doc_{past}\right) = \frac{doc_{new}.doc_{past}}{|doc_{new}||doc_{past}|} \quad (1)$$

At the conclusion of this step, we have identified the K most relevant source code files to the given change request. Now, we need to mine the candidate developers from the interaction histories of these files.

Fig. 2: A snippet of the bug #315184 interaction log entry from its interaction log file.

C. Mining Interaction Histories to Recommend Developers

The basic premise of *iHDev* is that the attachers who substantially interacted with the specific source code in the past are most likely the best candidate to assist with the issues/bugs associated with it. Our approach uses an *interaction log*, which was assembled from source code interactions submitted by attachers to the bug tracking system (e.g., *Bugzilla*). Interaction log entries include the dimensions: attacher, date, and path (e.g., files) involved in an interaction event. Interaction logs are not directly available from the issue/bug tracking systems. We engineered the interaction log to simplify the implementation of the mining component of *iHDev*. A notable side effect is that it provides an analogy to commit logs, which are a basis for several developer-recommendation approaches.

`Interaction Log:` An Interaction Log is a file that includes the interaction history and the attacher(s) who created the trace file for each bug in the issue tracking system. The specific steps for creating an interaction log are given below:

1) Extracting Interactions: We first need to identify the bug reports with the *mylyn-context.zip* attachment(s) because not all bug issues necessarily contain the interaction trace(s). We searched the *Bugzilla* issue tracking system and included bugs containing at least one *mylyn-context.zip* attachment.

2) Determining the list of attachers: All the trace files from the issue-tracking system are automatically downloaded to a user specified directory. A complete list of attachers for each trace file of every bug is then produced.

3) Creating Interaction log: The tool then takes the directory that contains the trace files as input and parses each trace file to create an interaction log. For each bug ID, this interaction log includes the attacher (from the list of attachers) and three of the eleven attributes from the trace file (see Figure 1): *EndDate, Kind*, and *StructureHandle*. There are several different kinds of interaction events (e.g., edit, manipulation, selection, and propagation); however, we consider only the edit interaction events because these events refer to interactions that resulted in a change to the source code file (even if that change was never committed), an action that often requires some level of knowledge of the source code. Other events such as navigation or selection do not imply explicit interaction or expertise and can therefore potentially introduce noise rather than provide additional useful information. Also, it is possible for one bug to have multiple trace files, each with a different attacher. Thus, for each bug in the interaction log file, multiple log entries may exist. Figure 2 shows a log entry for bug #315184 in the interaction log, which has only one trace file (and one attacher). For this bug, only one file has an edit interaction event (i.e.,*"AbstractTaskRepositoryPage.java"*).

The interaction log includes data such as the bug ID, attacher, the interacted source code, and the date on which the source code was interacted to fix this issue (see Figure 2 for an example). This data explains who interacted with the source code and when they did it. An attacher may contribute multiple interactions with the same file or multiple attachers may interact with the same file in different trace files. Therefore, interactions give an opportunity to analyze both the exclusive and shared contributions of attachers to a source code file.

`Developer Expertise Measures`: One measure of an attacher's contribution is the total number of interactions on source code performed in the past [11]. An attacher who contributed a larger number of interactions on a specific part of the source code than another attacher can be considered as more knowledgeable on those parts. Another consideration for attacher contributions is the workdays (i.e., activity) involved in the interactions that are attached as a trace file. The activity of a specific attacher is the percentage of their workdays over the total workdays of the system. Here, an attacher's workday is considered as a day (calendar date) on which they interacted with at least one part of the source code, because an attacher can have multiple interactions on a given workday. A system's workday is considered a day on which at least one part of the source code is interacted. A day on which no interactions exist is not considered a workday. These two measures give us two different views on attachers' development styles. Some may perform smaller interaction sessions and submit frequently in a workday (e.g., multiple attachments), while others may do it differently (e.g., single attachment). The third measure accounts for the recency of these interactions. We used these three measures, which were inspired by our previous work on commits [2], to determine the attachers that were more likely to be experts in a specific source code file, i.e., *attacher-interaction* map. The *attacher-interaction* map, *AI* for the attacher *a* and file *f* is given below:

$AI_{(a,f)} = \langle I_f, A_f, R_f \rangle$

- I_f is the number of interactions that include file *f* and are interacted by the attacher *a*.
- A_f is the number of workdays in the activity of attacher *a* with interactions that include the file *f*.
- R_f is the most recent workday in the activity of the attacher *a* with an interaction that includes the file *f*.

Similarly, the *file-interaction* map *FI* represents the interaction contribution to the file *f*, and is shown below:

$FI_{(f)} = \langle I'_f, A'_f, R'_f \rangle$, where

- I'_f is the number of interactions that include file *f*.
- A'_f is the total number of workdays in the activity of all attachers that include interactions with the file *f*.
- R'_f is the most recent workday with an interaction that includes the file *f*.

The measures I_f, A_f, and R_f are computed from the interaction log. More specifically, the dimensions attacher, date, and paths of the log entries are used in the computation. The dimension date is used to derive workdays or calendar days. The dimension attacher is used to derive the attacher information. The dimension path (StructureHandle) is used to derive the file information. The measures I'_f, A'_f, and R'_f are similarly computed. The log entries are readily available in the form of *XML* and straightforward *XPath* queries are formulated to compute the measures. The contribution or expertise factor, termed *xFactor*, for the attacher *a* and the file *f* is computed using the ratios of the *attacher–interaction* and *file–interaction* maps. The contribution factor, *xFactor*, is given below:

$$xFactor(a, f) = \frac{AI_{(a,f)}}{FI_{(f)}} \tag{2}$$

$$xFactor(a, f) = \begin{cases} \frac{I_f}{I'_f} + \frac{A_f}{A'_f} + \frac{1}{|R_f - R'_f|} & \text{if } |R_f - R'_f| \neq 0 \\ \frac{I_f}{I'_f} + \frac{A_f}{A'_f} + 1 & \text{if } |R_f - R'_f| = 0 \end{cases} \tag{3}$$

The *xFactor* score is computed for each of the relevant source code files to the given change request (see Section II-B). According to Equation 3, the maximum value of *xFactor* can be three because we have used three measures, each of which can have a maximum contribution ratio of 1.

`Recommending developers based on xFactor scores`: We now describe how the ranked-list of developers is obtained from all of the scored attachers of each relevant source code file to a given bug. From Section II-C, there is a one-to-many relationship between the source code files and attachers. That is, each file f_i may have multiple attachers; however, it is not necessary for all of the files to have the same number of attachers. For example, the file f_1 could have two attachers and the file f_2 could have three attachers. Each row in the matrix D_a (see Equation 4) gives the list of unique attachers for each relevant file f_i. D_{af_i} represents the set of attachers, with no duplication, for the file f_i, where $1 \leq i \leq n$ and n is the number of relevant files. a_{ij} is the j^{th} attacher in the file f_i with l unique attachers.

$$D_a = \begin{pmatrix} f_1 & D_{af_1} \\ f_2 & D_{af_2} \\ \vdots & \vdots \\ f_n & D_{af_n} \end{pmatrix} \quad D_{af_i} = \{\, a_{i1}\ a_{i2}\ \ldots\ a_{il} \,\} \tag{4}$$

Although, a single file does not have any duplicate attachers, two files may have common attachers. In Equation 5, D_{au} is the union of all unique attachers from all relevant files.

$$D_{au} = \bigcup_{i=1}^{n} D_{af_i} \tag{5}$$

$$Score(a) = \sum_{i=1}^{n} xFactor_i(a, f_i) \tag{6}$$

Each attacher a for a file f has the $xFactor$ score. To obtain the likelihood of the attacher a, i.e., $Score(a)$, to resolve the given change request, we sum $xFactor$ scores of the relevant files in which it appears (see Equation 6). That is, their overall expertise in all the files is computed collectively. The $Score(a)$ value is calculated for each unique attacher a in the set D_{au}.

In Equation 7, we have a set of candidate developers. The developers in this set are ranked based on their $Score(a)$ values. Once the developers are ranked in the descending order of their $Score(a)$ values, we have a ranked list of candidate developers. By using a cutoff value of m, we recommend the

TABLE I: The attachers extracted with *iHDev* from each of the top ten files relevant to Bug# 313712.

Files	**Ranked Attacher based on *xFactor* value**
.../TaskEditorNewCommentPart.java	Jingwen 'Owen':1.23, Frank Becker:1.06, Steffen Pingel:0.54, Jacek Jaroczynski:0.19, David Shepherd:0.07, David Green:0.06
.../CopyContextHandler.java	Shawn Minto:2.00, Steffen Pingel:0.61, Mik Kersten:0.42
.../NewAttachmentWizardDialog.java	Frank Becker:1.40, David Green:0.90, Steffen Pingel:0.60, Mik Kersten:0.30
.../Messages.java	....
.../WebBrowserDialog.java	....
.../BugzillaResponseDetailDialog.java	Frank Becker:3.00
.../UpdateAttachmentJob.java	Frank Becker:1.94, Robert Elves:1.40
.../TaskAttachmentPropertyTester.java	Philippe Marschall:1.63, Steffen Pingel:1.38
.../TaskEditorRichTextPart.java	Frank Becker:1.11, Jingwen 'Owen' Ou:0.90, Steffen Pingel:0.66, Thomas Ehrnhoefer:0.26, Jacek Jaroczynski:0.12, David Green:0.11, David Shepherd:0.03
.../BugzillaPlanningEditorPart.java	Steffen Pingel:1.85, Robert Elves:1.35

top m candidate developers, i.e., with top m $Score(a)$ values, from the ranked list obtained from the set DF.

$$DF = \{(a, Score(a)), \forall a \in D_{au}\} \quad (7)$$

This step concludes *iHDev* and we have the top m candidate developers recommended for the given change request.

D. An Example from Mylyn

Here, we demonstrate *iHDev* using an example from *Mylyn*. The change request of interest here is the bug #313712 *"attachment dialog does not resize when Advanced section is expanded"*. We first collected a snapshot of *Mylyn*'s source code prior to this bug fixed and then parsed it using the file-level granularity (i.e., each document is a file). After indexing with a machine learning technique, we obtained a corpus consisting of 1,825 documents and 201,554 words. A search query was then formulated using the bug's textual description, the result of which (i.e., top 10 relevant files) is summarized in Table I. These ($k = 10$) files are our starting point for *iHDev*. The correct developer who fixed this bug and committed the change is *Frank Becker*. In Table I, the third column shows a set of all attachers with the *xFactor* values for each file f_i.

In *iHDev*, we first obtained the set D_{au} from all of the attachers recommended for each relevant file f_i to the bug #313712 in Table I. The set D_{au} consists of 11 unique attachers. Because a developer could use different identities for the attacher and committer roles, we normalized them to a single identity, which was their full name. For each of the 11 unique attachers, the $Score$ value is calculated according to Equation 6. Table II shows the top five $Score$ values and the corresponding attachers, i.e., $m = 5$. *Frank Becker* has the highest score in the set DF (a value of 8.51), so he is the first recommended developer. For the remaining attachers, the value of the function $Score$ is less than *Frank Becker*'s score, so they all have a rank greater than 1.

Table III shows the results for the approaches *iHDev*, *xFinder*, *xFinder'* and *iMacPro* (described in sections III-A) for $m = 5$. Clearly, the best result is for *iHDev*, as it recommends *Frank Becker* (the correct developer who fixed bug #313712) in the first position, whereas *xFinder*, *xFinder'*, and *iMacPro* recommend *Frank Becker* in the third, fourth, and third position respectively. *iHDev* outperforms the others with respect to recall@1 and recall@2 values. At recall@5, all the approaches would be equivalent; however, *iHDev* provides the best ranked position (i.e., the reciprocal ranked value).

TABLE II: Top five attachers (developers) recommended to resolve bug #313712 by *iHDev*.

Attacher	***Score***	**Rank**
Frank Becker	8.51	1
Steffen Pingel	5.64	2
Robert Elves	2.75	3
Jingwen 'Owen' Ou	2.13	4
Shawn Minto	2.0	5

TABLE III: Top five recommended developers and their associated ranks for the compared approaches. *iHDev*, *xFinder*, *xFinder'* and *iMacPro*.

Approach	**Top five recommended developers in ranked order**
iHDev	Frank Becker ①, Steffen Pingel ②, Robert Elves ③, Jingwen 'Owen' Ou ④, Shawn Minto ⑤
xFinder	Steffen Pingel ①, Robert Elves ②, Mik Kersten ②, Frank Becker ③
xFinder'	Steffen Pingel ①, Mik Kersten ②, Robert Elves ③, Frank Becker ④, Shawn Minto ⑤
iMacPro	Steffen Pingel ①, Shawn Minto ②, Frank Becker ③

III. Case Study

The purpose of this study was to investigate how well our *iHDev* approach recommends expert developers to assist with incoming change requests. We also compared our approach with two previously published approaches. The first approach, *xFinder* [2], is based on the mining of commit logs. The second approach, *iMacPro* [6], uses the authorship information and maintainers of the relevant change prone source code to recommend developers. We used these two approaches for comparison because they require information from the commit repository. *iHDev* uses interaction history of source code. Therefore, this part of the study would allow us to compare interactions and commits with respect to the developer recommendation task. We addressed the following research question **RQ**: How do *iHDev* (trained from the interaction history) compare with *xFinder*, *xFinder'*, and *iMacPro* (trained from the commit history) in recommending developers?

A. Compared Approaches: xFinder, xFinder′, and iMacPro

The *xFinder* approach uses the source code commit history to recommend developers to assist with a given change request. The first step is finding the relevant source code files for the change request, similar to *iHDev* (see Section II-B). The commit (and not interaction) histories of these files are analyzed to recommend a ranked list of developers. *xFinder* uses the frequency count of the developers in the relevant files for ranking purposes. For example, if a developer occurs in three relevant files, its score is assigned a value of 3, and is ranked above another developer that occurs in two relevant files (with a score of 2). If multiple developers have the same score, they are given the same rank (e.g., two developers are ranked second in Table III). *iHDev* uses a different ranking mechanism. We replaced *xFinder*'s ranking algorithm with that of *iHDev*, which is based on the sum of $xFactor$ scores. This modified *xFinder* approach is termed *xFinder′*. *xFinder′* allows us to compare the core of two approaches (i.e., interactions and commits) by neutralizing the variability in their rankings.

The *iMacPro* approach uses the source code authorship and commit history to assign incoming change requests to the appropriate developers. The first step is finding the relevant source code files for the change request, similar to *iHDev* (see Section II-B). These source code units are then ranked based on their change proneness. Change proneness of each source code entity is derived from its commit history. The developers who authored and maintained these source code files are discovered and combined. Finally, a final ranked list of developers for the given change request is recommended. Hossen et al. [6] showed that *iMacPro* outperformed the *iA* approach [3], which was shown to perform equivalent to or better than the approach of Anvik et al. [1].

B. Subject Software Systems

We focused on the *Mylyn* and *Eclipse Project*.

1) Mylyn: Mylyn contains about 4 years of interaction data and is an Eclipse Foundation project with the largest number of interaction history attachments. It is mandatory for *Mylyn* project committers to use the *Mylyn* plug-in. Commit history started 2 years prior to that of interaction, and commits to the *Mylyn* CVS repository terminated on July 01, 2011. To get both interaction and commit histories within the same period, we considered the history between June 18, 2007 (the first day of an interaction history attachment) and July 01, 2011 (the last day of a commit to the *Mylyn* CVS repository). Doing so ascertained that none of the approaches had any particular advantage or disadvantage over the others due to the lack of history for training. The *Mylyn* project consists of 2272 bug issues containing 3282 trace files for a total of 276390 interaction logs for interaction events that lead to edit of the interacted source code. The *Mylyn* project consists of 5093 revisions, out of which 3727 revisions contained a change to at least one Java file and for 3536 revisions there exists at least one trace file in the interaction history.

2) Eclipse Project: Eclipse Project contains 6 different sub projects: *e4*, *Incubator*, *JDT*, *Orion*, *PDE* and *Platform*[2]. *Eclipse Project* contains about 7 years of interacted data from July 2007 to May 2014. It is not mandatory for the *Eclipse Project* committers to use the *Mylyn* plug-in. Therefore, it is not surprising that the number of issues that contain interactions is less than those in the *Mylyn* project. The commit history for *Eclipse Project* started in April 2001, 6 years prior to that of interaction, and commits to the *Eclipse Project* CVS repository terminated on November 05, 2012. To get both interaction and commit histories within the same period, we considered the history between July 25, 2007 (the first day of an interaction history attachment) and November 05, 2012 (the last day of commits/revisions to the *Eclipse Project* CVS repository). During this history the *Eclipse Project* consists of 700 bug issues containing 897 trace files for a total of 95834 interaction logs for interaction events that lead to edit of the interacted source code. It consists of 52126 revisions, out of which 35290 revisions contained a change to at least one Java file and for 861 revisions there exists at least one trace file in the interaction history.

Both *xFinder* and *iMacPro* need the commit histories of the *Mylyn* and *Eclipse Project* open source systems because they need files that have been changed together in a single commit operation. SVN preserves the atomicity of commit operations; however, older versions of CVS did not. Subversion assigns a new "revision" number to the entire repository structure after each commit. The "older" CVS repositories were converted to SVN repositories using the CVS2SVN tool, which has been used in popular projects such as *gcc3*. For the datasets considered in our study, *Mylyn* and *Eclipse Project* had their commit histories in CVS depositories. Therefore, we converted the CVS repositories to SVN repositories.

C. Benchmarks: Training and Testing Datasets

For *Mylyn* and *Eclipse Project*, 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, their historical interactions in the bug tracking system, and their historical changes recorded in version-control repositories. Interaction trace files were used to find the interaction events related to each bug and the attacher who created the trace file was used as the attacher in our interaction log. For tracing each *Bug ID* in the Subversion (SVN) repository commit logs, 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 who contributed changes to the change requests (i.e., gold set). These developers were then used to form our actual developer set for evaluation. A vast majority of change requests are handled by a single developer (92% in *Mylyn* and 97% in *Eclipse Project*). Table IV shows the frequency distributions of developers resolving issues in the

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

TABLE IV: The frequency distributions of developers resolving issues in the benchmarks for *Mylyn* and *Eclipse Project*.

System	Frequency distribution			Total Issues
	# 1	# 2	# 3	
Mylyn	277	21	3	301
Eclipse Project	70	0	2	72

benchmarks for *Mylyn* and *Eclipse Project*. For example, 277 issues were resolved by a single developer in *Mylyn* and none of the issues were resolved by two developers in *Eclipse Project*. Our technique operates at the change request level, so we also needed input queries to test. We considered all of the bugs that have at least one associated Id in the commit messages and traces files. We created the final training corpus from all of the source code files in the last revision before the bug issues in our benchmark were fixed. We split the benchmark into training and testing sets. We picked June 18, 2007 to March 16, 2010 as our training set and March 16, 2010 to July 01, 2011 as our testing set for *Mylyn*. The testing set period contains 600 different revisions and 301 different issues/bugs. Similarly We picked July 25, 2007 to March 01, 2010 as our training set and March 01, 2010 to November 05, 2012 as our testing set for *Eclipse Project*. The testing set contains 140 different revisions and 72 different issues/bugs. Our benchmarks are available online[3]. The experiment was run for $m = 1$, $m = 2$, $m = 3$, and $m = 5$, where m is the number of recommended developers. We considered the top *ten* relevant files from the machine learning step.

D. Metrics and Statistical Analyses

We evaluated the accuracy of each of the approaches for all of the bug issues in our testing set using the Recall and Mean Reciprocal Rank (MRR) metrics used in previous work [1], [3], [5]. For each bug *b*, in a set of bugs B of size n, in the benchmark of a system and a m number of recommended developers, the formula for the recall@m is given below:

$$recall@m = \frac{\mid RD(b) \cap AD(b) \mid}{\mid AD(b) \mid} \tag{8}$$

where *RD(b)* and *AD(b)* are the recommended developer by the approach and the actual developer who resolved the issue for the bug *b* respectively. This metric is computed for recommendation lists of developers with different sizes (e.g., $m = 1$, $m = 2$, $m = 3$, and $m = 5$ developers).

Increasing the value of m could lead to a higher recall value (and typically does); however, it may come at the cost of an increased effort in examining the potential noise (false positives). Over 90% of the cases in our benchmarks have only a single correct developer. In such a scenario, each increment to m in pursuit of a correct developer could add drastically to the proportion of false positives. Therefore, a traditional metric of the likes of precision is not a suitable fit for our context. For example, if a correct developer is found at $m = 5$, the possible precision value is in the range [0.2, 1.0] for the rank positions [5, 1], typically around the lower bound of 0.2. Note that the precision metric is also agnostic to the rank positions of recommendations. Therefore, for a cutoff value of m, it would produce the same value for two approaches presenting the same number of correct answers. For $m = 5$, two approaches presenting a single correct answer at the positions 1 and 5 respectively will have the same precision value of 0.2. Nonetheless, a complimentary measure to recall is also needed to assess the potential effort in addressing noise (false positives). We focused on evaluating the ranked position of the correct developer for each bug in each benchmark from a cumulative perspective regardless of the cutoff point m.

Mean Reciprocal Rank (MRR) is one such measure that can be used for evaluating any process that produces a list of possible responses to a sample of queries, ordered by probability of correctness. This metric has been used in previous work [12]. The reciprocal rank of a query response is the multiplicative inverse of the rank of the first correct answer. Intuitively, the lower the value (between 0 and 1), the farther down the list, examining incorrect responses, one would have to search to find a correct response.

$$MRR = \frac{1}{\mid n \mid} \sum_{i=1}^{\mid n \mid} \frac{1}{rank_i} \tag{9}$$

Here, the reciprocal rank for a query (bug) is the reciprocal of the position of the correct developer in the returned ranked list of developers ($rank_i$) and n is the total number of bugs in our benchmark. When the correct developer for a bug is not recommended at all, we consider its inverse rank to be a zero. When there are multiple correct developers (which are a very few cases in our benchmark), we consider the highest/first ranked position. The higher the value of MRR, the better it speaks of the potential effort spent in noise. For example, an MRR value of 0.5 suggests that the approach typically produces the correct answer at the 2nd rank. **Overall, our main research hypothesis is that *iHDev* will outperform the subjected competitors in our study in terms of recall without incurring additional costs in terms of MRR**.

We applied the One Way ANOVA test to validate whether there was a statistically significant difference with $\alpha = 0.05$ between the results of both recall and MRR values. For MRR, we considered the ranks of correct answers of the approaches for each bug (data point). We used this non-parametric test because we did not assume normality in the distributions of the recall results. The 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 can be easily derived from the respective null hypotheses):

- **H-1:** There is no statistically significant difference between the recall@m values of *iHDev* and each of *xFinder*, *xFinder′*, and *iMacPro*.
- **H-2:** There is no statistically significant difference between the MRR values of *iHDev* and each of *xFinder*, *xFinder′*, and *iMacPro*.

E. Results

The recall@1, recall@2, recall@3, and recall@5 values for each of the compared approaches for *Mylyn* and *Eclipse Project* were calculated from the established benchmarks.

[3]http://serl.cs.wichita.edu/svn/projects/dev-rec-interactions/trunk/MSR2015/data/Benchmark

TABLE V: Average of recall @1, 2, 3 and 5 values of the approaches *iHDev*, *xFinder*, *xFinder′*, and *iMacPro* measured on the *Mylyn* and *Eclipse Project* benchmarks.

m	**Recall@m**				***iHDev* vs *xFinder***			***iHDev* vs *xFinder′***			***iHDev* vs *iMacPro***		
	iHDev	*xFinder*	*xFinder′*	*iMacPro*	**Gain%**	**Pvalue**	**Adv**	**Gain%**	**Pvalue**	**Adv**	**Gain%**	**Pvalue**	**Adv**
Mylyn													
1	0.50	0.45	0.52	0.50	11.11	0.22	*None*	-3.84	0.58	*None*	0	0.9	*None*
2	0.71	0.63	0.59	0.63	12.69	0.02	*iHDev*	20.33	0.001	*iHDev*	12.69	0.03	*iHDev*
3	0.79	0.68	0.69	0.72	16.17	0.001	*iHDev*	14.49	0.004	*iHDev*	9.72	0.05	*iHDev*
5	0.86	0.69	0.81	0.76	24.63	≤0.001	*iHDev*	6.17	0.08	*None*	13.15	0.0008	*iHDev*
Eclipse Project													
1	0.25	0.12	0.12	0.11	108.33	0.03	*iHDev*	108.33	0.03	*iHDev*	127.27	0.02	*iHDev*
2	0.37	0.20	0.17	0.18	85.00	0.03	*iHDev*	117.64	0.007	*iHDev*	105.55	0.01	*iHDev*
3	0.41	0.20	0.22	0.22	105.00	0.005	*iHDev*	86.36	0.01	*iHDev*	86.36	0.01	*iHDev*
5	0.45	0.20	0.23	0.23	125.00	0.001	*iHDev*	95.65	0.004	*iHDev*	95.65	0.003	*iHDev*

TABLE VI: Mean Reciprocal Rank of the approaches *iHDev*, *xFinder*, *xFinder′*, and *iMacPro* measured on the *Mylyn* and *Eclipse Project* benchmarks.

System	**MRR**				***iHDev* vs *xFinder***			***iHDev* vs *xFinder′***			***iHDev* vs *iMacPro***		
	iHDev	*xFinder*	*xFinder′*	*iMacPro*	**Gain%**	**Pvalue**	**Adv**	**Gain%**	**Pvalue**	**Adv**	**Gain%**	**Pvalue**	**Adv**
Mylyn	0.66	0.56	0.62	0.64	17.85	0.003	*iHDev*	6.45	0.22	*None*	3.12	0.4	*None*
Eclipse	0.34	0.16	0.16	0.17	112.5	0.005	*iHDev*	112.5	0.007	*iHDev*	100	0.01	*iHDev*

Table V shows the $recall@m$ values of *Mylyn* and *Eclipse Project* for all the compared approaches (see the Recall@m column). The MRR values for each of the compared approaches for *Mylyn* and *Eclipse Project* were then calculated. Table VI shows the MRR values of *Mylyn* and *Eclipse Project* for all approaches (see the MRR column).

We computed the recall gain of *iHDev* over another compared approach (i.e., Y) using the following formula:

$$GainR@m_{iHDev-Y} = \frac{recall@m_{iHDev} - recall@m_Y}{recall@m_Y} \times 100 \quad (10)$$

The MRR column in Table VI shows MRR values of the compared approaches.

We computed the MRR gain of *iHDev* over another compared approach (i.e., Y) using the following formula:

$$GainMRR_{iHDev-Y} = \frac{MRR_{iHDev} - MRR_Y}{MRR_Y} \times 100 \quad (11)$$

To answer the research question RQ, we compared the recall values of *iHDev*, *xFinder*, *xFinder′*, and *iMacPro* for $m = 1$, $m = 2$, $m = 3$, and $m = 5$. We computed the recall gain of *iHDev* over Y in {*xFinder*, *xFinder′*, and *iMacPro*} using Equation 10. Similarly, we compared MRR values of *iHDev*, *xFinder*, *xFinder′*, and *iMacPro*. We computed the MRR gain of *iHDev* over Y in {*xFinder*, *xFinder′*, and *iMacPro*} using Equation 11. Table V and Table VI show the recall and MRR results respectively. The **Gain %** columns in Table V show the recall gains of *iHDev* over each compared approach for the different m values. The **Gain %** columns in Table VI show the MRR gain of *iHDev* over each compared approach. The **Pvalue** columns in Table V shows the p-values from applying the One Way ANOVA test on the recall values for $m = 1$, $m = 2$, $m = 3$, and $m = 5$. The **Pvalue** columns in Table VI shows the p-values from applying the One Way ANOVA test on the reciprocal rank values. The **Advantage** columns show the approach that had a statistically significant gain over the other. In cases neither did, **None** is shown.

For each pair of competing approaches, there were eight comparison points in terms of recall values (four each for *Mylyn* and *Eclipse Project*). Overall, out of eight comparison cases between *iHDev* and *xFinder* for recall values, *iHDev* was advantageous over *xFinder* in seven of them; the remaining one being a statistical tie. Out of eight comparison cases between *iHDev* and *xFinder′* for recall values, *iHDev* was advantageous in six of them; the remaining two being a statistical tie. Out of eight comparison cases between *iHDev* and *iMacPro* for recall values, *iHDev* was advantageous in seven of them; the remaining one being a statistical tie. Despite a few observations of negative gains, there was not even a single case in which other approaches was statistically advantageous over *iHDev* in terms of recall. Therefore, we reject the hypothesis H_1.

For each pair of competing approaches, there were two comparison points in terms of MRR values (one each for *Mylyn* and *Eclipse Project*). Overall, *iHDev* was advantageous in both cases of comparison between *iHDev* and *xFinder*. *iHDev* was advantageous in one case each for comparisons between *iHDev* and *xFinder′*, and between *iHDev* and *iMacPro*. The remaining two cases were a statistical tie. There were no cases in which the other approaches were statistically advantageous over *iHDev* in terms of MRR. Therefore, we reject the hypothesis H_2.

In summary, the overall results suggest that *iHDev* generally performs better than *xFinder*, *xFinder′*, and *iMacPro* in terms of both recall and MRR. Using the interaction history typically leads to improvements in accuracy. Not only does it identify the correct developers more often (as evident by the significant recall gains or no loss), but also at a high enough position in

the list of recommended candidates (as evident by the significant MRR gains or no loss). For example, *iHDev* recorded recall gains over *xFinder* in the range [85% , 125%] for *Eclipse Project*. Also, on average, the correct developer would appear at the 3rd position (MRR=0.34) for *iHDev* recommendations, whereas, in *xFinder* this value would be at the 6th position (MRR=0.16). Thus, the MRR gain of over 112%. ***iHDev* takes us a step forward toward achieving the ideal goal of a recommender that not only always identifies the correct developers, but also puts them in the top positions.**

F. Discussion

We discuss a few qualitative points that would help understand the rationale behind the improved performance with using interactions in *iHDev*.

Multiple Attempts at Resolution. Given the nature of OSS, there are often multiple attempts at resolving a given change request. For example, multiple (a few incomplete or incorrect) fixes are attempted by perhaps multiple developers. In the end, only a complete and correct resolution is accepted and/or merged into the source code repository (i.e., the main development trunk or branch). The interaction history records these attempts; however, the commit history only records the final outcome (i.e., only the things committed), if any. Gousios et al. [13] observed that in GitHub some issues receive multiple pull-requests with solutions (all representing interaction and competence), but not all are accepted and merged. Our results show that past experiences (including failures) are important ingredients in shaping developer expertise. Interactions offer a valuable insight into these micro-level developers' activities in building their expertise, whereas, the commit repositories would largely miss out on them. For example, the file *UpdateAttachmentJob.java* has 458 edit events, but it has only 6 commits. Additionally, the developer *Frank Becker* contributed 50 edit events, but had only one commit. The interaction history shows that he attempted resolutions for 6 bugs, but, the commit shows that he contributed only one bug fix. This example suggests there is quite a disparity between the micro and macro level perspectives of contributions.

Resolutions and Peer Review. The practice of peer review of proposed resolutions/patches exist in both *Mylyn* and *Eclipse Project*. We observed cases in which resolutions (even those that were correct in the sense of a technical fix for the problem at hand) were not merged to the source code repository (e.g., potential conflicts with other code elements or patches did not make it in time for the review process to go through or were out of luck as they arrived after something else was already accepted) or were revised and then merged (e.g., a few changes to a few files revised or not needed). The interaction history captures the resolutions while the commit history misses out on these alternative solutions.

Interactions come first, Commits later. In the typical workflow, interactions come first and commits later. Interactions are available much earlier than commits to mine and integrate the results. Our datasets show that for a given time period, there are more interactions than commits. Therefore, using interaction data may reduce the latency in building and deploying actionable models on a project, as the training data would become available sooner.

Self-Serving Benefits. We believe that the potential benefits shown to developers by converting the interactions into actionable support for their routine tasks could serve as a motivating factor in using *Mylyn* type activity loggers. That is, developers would see the value in logging their activities now, so that they could benefit for concrete tasks in the future.

Task Applicability. Our work shows the potential value of interactions in improving the developer recommendation tasks; however, they could be use for other software maintenance and evolution tasks that have typically relied on commit histories. Previously, we used interactions for impact analysis [7], [8].

In summary, our findings highlight many aspects that the developer recommendation methodology based on the commit history may not capture, but interactions do.

IV. Threats to Validity

We discuss internal, construct, and external threats to validity of the results of our empirical study.

Accuracy measures and correctness of developer recommendations: We used the widely used metric recall in our study. We also calculated mean reciprocal rank. 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. Nonetheless, our benchmark can be viewed as conservative bounds.

Accuracy@k [14]–[16]generally considers only one correct answer to claim 100%. In our study, over 90% of cases had only one correct answer. Therefore, the differences in recall@k and accuracy@k values are negligibly small. Although, (as discussed in Section III-D) we deemed the precision metric to be unsuitable for our technique due to the ordered nature of responses, we summarize the results. In case of *Mylyn* and *xFinder* precision gain ranges between 12.76% and 136.36% and out of eight comparisons with ANOVA, *iHDev* was advantageous in four of them. In case of *Mylyn* and *xFinder′* precision gain ranges between 12% and 137.50 % and out of eight comparisons with ANOVA, *iHDev* was advantageous in six of them. In case of *Mylyn* and *iMacPro* precision gain ranges between 6% and 136.36 % and out of eight comparison with ANOVA, *iHDev* was advantageous in seven of them. Furthermore, should there be a larger number of cases with multiple correct answers, a metric such as Mean Average Precision (MAP) could be employed.

Machine Learning-based matching of change requests to relevant files: KNN sometimes returned classes (i.e., files) that were not found in the commits related to the bug fixes or change request implementations. 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).

Narrow Ground Truth: Our ground truth included only the developers who eventually resolved the change request. It is likely that other developers are equally capable of resolving

the same requests. Identifying them is not obvious from repositories and would require a quantitative study.

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

Incomplete or Missing Interaction History: Although, a common period was considered for extracting the interaction and commit datasets, the number of commits is significantly higher than the number of interaction transactions. This difference is not necessarily the result of a single task defined for multiple commits, because there are many cases in which committed files were never part of one of the interactions.

Explicit Bug ID Linkage: We considered interactions and commits to be related if there was an explicit bug ID mentioned in them. Implicit relationships were not considered.

Two Systems Considered: Due to the limited availability of Mylyn interaction histories, our study was performed on only two systems written in Java. *Mylyn* and *Eclipse Project* were the largest available datasets within the Eclipse Foundation. Nonetheless, this fact may limit the generality of our results.

V. Related Work

Our intent is not to exhaustively discuss every single work, but to briefly discuss a few representatives in the context of developer recommendation and interactions.

Zou et al. [17] used the interaction history to identify evolutionary information about a development process (e.g., restructuring is more costly than other maintenance activities). Rastkar et al. [18] report on an investigation which considered how bug reports that are considered similar based on the changeset information compare to bug reports that are considered similar based on interaction information. Robbes et al. [19] developed an incremental change based repository by retrieving the program information from an IDE (which includes more information about the evolution of a system than traditional SCM) to identify a refactoring event. Parnin and Gorg [20] identified relevant methods for the current task by using programmers' interactions with an IDE. Schneider et al. [21] presented a visual tool for mining local interaction histories to help address some of the awareness problems experienced in distributed software development projects.

Minto and Murphy [22] developed a tool called Emergent Expertise Locator (EEL), which is based on the framework of Cataldo et al. [23] to compute coordination requirements between documents. Another tool to identify developers with the desired expertise is Expertise Browser (ExB) [11]. Anvik and Murphy [24] conducted an empirical evaluation of two techniques for identifying expert developers. Developers acquire expertise as they work on specific parts of a system. Tamrawi et al. [4] used fuzzy-sets to 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. An approach based on a machine learning technique is used to automatically assign a bug report to a developer [1]. Another approach to facilitate bug triaging uses a graph model based on Markov chains, which captures the bug reassignment history [25]. Matter et al. [26] used the similarity of textual terms between a given bug report of interest and source code changes (e.g., word frequencies of the *diff* given changes from source code repositories). Fritz et al. [27] introduced the degree-of-knowledge model that computes a real value for each source code element based on both authorship and interaction information. Their expertise model was not applied to, and evaluated on, the developer recommendation problem.

Rahman and Devanbu [28] study the impact of authorship on code quality. They conclude that authors with specialized experience for a file are more important than those with general expertise. Bird et al. [29] 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. Bird et al. [30] analyzed the communication and co-ordination activities of the participants by mining email archives. Ma et al. [31] proposed a technique that uses implementation expertise (i.e., developers usage of API methods) to identify developers. Weissgerber et al. [32] depicts the relationship between the lifetime of the project and the number of files each author updates by analyzing and visualizing the check-in information for open source projects. German [33] provided a visualization to show which developers tend to modify certain files by studying the modification records (MRs) of CVS logs. Fischer et al. [34] analyzed related bug report data for tracking features in software. Bortis et al. [35] 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. [36] proposed an approach for bug report assignment based on the predicted location (in the source code) of the bug. Begel et al. [37] conducted a survey of inter-team coordination needs and presented a flexible *Codebook* framework that can address most of those needs. Robbes et al. [38] used the interaction data from *Mylyn* to evaluate developer-expertise metrics based on time, which is the start time of a *Mylyn*-interaction session.

VI. Conclusions and Future Work

We presented the *iHDev* approach to recommend developers who are most likely to implement incoming change requests. *iHDev* utilizes the archives of developers' interactions within an integrated development environment (e.g., software entities viewed or modified) associated with past change requests. Such use of interaction histories in conjunction with machine learning techniques to recommend developers was not investigated previously. Moreover, an empirical study on the open source systems showed that *iHDev* can outperform two previous approaches with statistically significant recall and rank gains. In the future, we plan to conduct additional empirical studies to further validate *iHDev*. We would include approaches based on the textual similarity of an incoming change request to previously resolved ones in the comparison study. Furthermore, we will investigate integration of other sources of information (e.g., bug, commit, and code review histories), which could further improve its effectiveness.

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*, ICSE '06, pp. 361–370, 2006.

[2] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Hammad, "Assigning change requests to software developers," *Journal of Software: Evolution and Process*, vol. 24, no. 1, pp. 3–33, 2012.

[3] 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, 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*, ESEC/FSE '11, pp. 365–375, 2011.

[5] 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, 2013.

[6] K. Hossen, H. Kagdi, and D. Poshyvanyk, "Amalgamating source code authors, maintainers, and change proneness to triage change requests," in *Proceedings of the 22th IEEE International Conference on Program Comprehension, ICPC*, 2014.

[7] F. Bantelay, M. Zanjani, and H. Kagdi, "Comparing and combining evolutionary couplings from interactions and commits," in *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*, pp. 311–320, Oct 2013.

[8] M. B. Zanjani, G. Swartzendruber, and H. Kagdi, "Impact analysis of change requests on source code based on interaction and commit histories," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, (New York, NY, USA), pp. 162–171, ACM, 2014.

[9] G. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the elipse ide?," *IEEE Transactions on Software Engineering*, vol. 23, pp. 76–83, July 2006.

[10] M.-L. Zhang and Z.-H. Zhou, "Ml-knn: A lazy learning approach to multi-label learning," *Pattern Recogn.*, vol. 40, pp. 2038–2048, July 2007.

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

[12] S. Wang and D. Lo, "Version history, similar report, and structure: Putting them together for improved bug localization," in *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, (New York, NY, USA), pp. 53–63, ACM, 2014.

[13] G. Gousios, M. Pinzger, and A. van Deursen, "An exploratory study of the pull-based software development model.," in *ICSE*, pp. 345–355, 2014.

[14] J. Xuan, H. Jiang, Z. Ren, and W. Zou, "Developer prioritization in bug repositories," in *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, (Piscataway, NJ, USA), pp. 25–35, IEEE Press, 2012.

[15] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered harmful ..... really?," in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pp. 337–345, Sept 2008.

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

[17] L. Zou, M. Godfrey, and A. Hassan, "Detecting interaction coupling from task interaction histories," in *Proceedings of the 15th IEEE International Conference on Program Comprehension,ICPC '07*, pp. 135–144, June 2007.

[18] S. Rastkar and G. Murphy, "On what basis to recommend: Changesets or interactions?," in *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories, MSR '09*, pp. 155–158, May 2009.

[19] R. Robbes, "Mining a change-based software repository," in *Proceedings of the Fourth International Workshop on Mining Software Repositories, 2007. ICSE Workshops MSR '07.*, pp. 15–15, May 2007.

[20] C. Parnin and C. Gorg, "Building usage contexts during program comprehension," in *Proceedings of the 14th IEEE International Conference on Program Comprehension, ICPC*, pp. 13–22, 2006.

[21] K. Schneider, C. Gutwin, R. Penner, and D. Paquette, "Mining a software developer's local interaction history," in *Proceedings of the IEEE International Conference on Software Engineering Workshop on Mining Software Repositories,*, 2004.

[22] S. Minto and G. Murphy, "Recommending emergent teams," in *proceedings of fourth International Workshop on Mining Software Repositories, 2007. ICSE Workshops MSR '07.*, pp. 5–5, 2007.

[23] 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*, CSCW '06, pp. 353–362, 2006.

[24] J. Anvik and G. 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, 2007.

[25] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, (New York, NY, USA), pp. 111–120, ACM, 2009.

[26] 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, 2009.

[27] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill, "A degree-of-knowledge model to capture source code familiarity," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, (New York, NY, USA), pp. 385–394, ACM, 2010.

[28] F. Rahman and P. Devanbu, "Ownership, experience and defects: A fine-grained study of authorship," in *proceedings of 33rd International Conference on Software Engineering*, ICSE '11, pp. 491–500, 2011.

[29] 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*, ESEC/FSE '11, pp. 4–14, 2011.

[30] 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*, MSR '06, pp. 137–143, 2006.

[31] 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, 2009.

[32] 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, 2007.

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

[34] 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, 2003.

[35] G. Bortis and A. v. d. Hoek, "Porchlight: A tag-based approach to bug triaging," in *proceedings of 2013 International Conference on Software Engineering*, ICSE '13, pp. 342–351, IEEE Press, 2013.

[36] 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, 2013.

[37] 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 - Volume 1*, ICSE '10, pp. 125–134, 2010.

[38] R. Robbes and D. Röthlisberger, "Using developer interaction data to compare expertise metrics," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, (Piscataway, NJ, USA), pp. 297–300, IEEE Press, 2013.