

Who Should Review This Change?

Putting Text and File Location Analyses Together for More Accurate Recommendations

Xin Xia*, David Lo[†], Xinyu Wang^{*‡}, and Xiaohu Yang*

*College of Computer Science and Technology, Zhejiang University, Hangzhou, China

[†]School of Information Systems, Singapore Management University, Singapore

xxia@zju.edu.cn, davidlo@smu.edu.sg, {wangxinyu, yangxh}@zju.edu.cn

Abstract—Software code review is a process of developers inspecting new code changes made by others, to evaluate their quality and identify and fix defects, before integrating them to the main branch of a version control system. Modern Code Review (MCR), a lightweight and tool-based variant of conventional code review, is widely adopted in both open source and proprietary software projects. One challenge that impacts MCR is the assignment of appropriate developers to review a code change. Considering that there could be hundreds of potential code reviewers in a software project, picking suitable reviewers is not a straightforward task. A prior study by Thongtanunam et al. showed that the difficulty in selecting suitable reviewers may delay the review process by an average of 12 days.

In this paper, to address the challenge of assigning suitable reviewers to changes, we propose a hybrid and incremental approach TIE which utilizes the advantages of both Text mIning and a file location-based approach. To do this, TIE integrates an incremental text mining model which analyzes the textual contents in a review request, and a similarity model which measures the similarity of changed file paths and reviewed file paths. We perform a large-scale experiment on four open source projects, namely **Android, OpenStack, QT, and LibreOffice**, containing a total of 42,045 reviews. The experimental results show that on average TIE can achieve top-1, top-5, and top-10 accuracies, and Mean Reciprocal Rank (MRR) of 0.52, 0.79, 0.85, and 0.64 for the four projects, which improves the state-of-the-art approach REVFinder, proposed by Thongtanunam et al., by 61%, 23%, 8%, and 37%, respectively.

Index Terms—Modern Code Review, Recommendation System, Text Mining, Path Similarity

I. INTRODUCTION

Software code review has been the best practice in both open source and industrial software projects [11]. It is the process where developers inspect code changes made by others to evaluate the quality of the changes and identify and fix defects before integration. Previous studies show that formal inspections of code with in-person meeting can help reduce the number of post-release defects [11], and improve the overall quality of software systems [5], [7]. Unfortunately, the cumbersome and time-consuming nature of conventional formal code inspection impedes its adoption in practice [25], [31].

Modern Code Review (MCR), an informal, lightweight, and tool-based variant of conventional formal code review, has been widely used in companies such as Microsoft, Google, Facebook, and many open source software projects [8]. In

MCR, typically a developer submits a code change (aka. patch) to a code review system, e.g., Gerrit¹, and recommends a set of developers to review the change. Then, the reviewers would discuss the change, provide comments, and suggest fixes to the change. Next, the developer improves the change according to the comments. The code change would be integrated into the main branch of the version control system when the reviewers approve it.

To effectively speed up the code review process, the developer should find appropriate code-reviewers who are expert in the related source code, and are able to inspect the code well to find bugs (if they exist) [7], [23], [28]. Thongtanunam et al. found that 4% - 30% of the reviews suffer from a code-reviewer assignment problem, and these reviews require, on average, an additional of 12 days to get completed [28]. Moreover, considering the large amount of review requests submitted daily (e.g., there are 74.4 review requests submitted daily to QT's Gerrit from 2011 - 2012), manually assigning appropriate code-reviewers can be tedious and time-consuming. Thus, an automated code-reviewer recommendation tool could help to relieve developer workload and speed up the code review process.

In this paper, we propose a hybrid and incremental approach TIE which utilizes the advantages of both Text mIning and a file location-based approach to address the code-reviewer recommendation problem. A typical review has two important fields, i.e., the description field which describes the details of the change (i.e., patch), and the file location field which records the paths of modified files, and TIE effectively makes use of both fields. TIE first builds an incremental text mining model by analyzing the textual contents in the description field of the reviews, and a similarity recommendation model by measuring the similarity of the paths of the changed files and paths of files historically reviewed by various developers. The similarity of a pair of paths is computed based on the number of common directory and file names between them. Next, these two models are blended to achieve a better performance. TIE is an incremental approach, which can be updated with new training data as new code changes are reviewed by developers.

Our goal is to improve the state-of-the-art approach REVFinder which was recently proposed by Thongtanunam et al. [28]. REVFinder uses the file paths in the reviews

[‡]Corresponding author.

¹<https://code.google.com/p/gerrit/>

to recommend code reviewers. It measures the similarity of two paths by using four string comparison techniques, i.e., longest common prefix, longest common suffix, longest common substring, and longest common subsequence. TIE’s similarity model also uses the file paths, however it measures the similarity of two paths in a different way, i.e., by measuring the number of common directory and file names between them. Different from REVFinder, TIE also has a text mining model that considers the textual contents in the description field of review requests to recommend reviewers. Furthermore, TIE merges the text mining and similarity models to achieve better performance.

We evaluate our approach on 4 datasets from different software communities: Android [1], OpenStack [3], QT [4], and LibreOffice [2]. In total, we analyze 42,045 code reviews. We measure the performance of our approach and a competing baseline in terms of top-1, top-5, and top-10 recommendation accuracies, and Mean Reciprocal Rank (MRR) [9]. The experimental results show that TIE achieves average top-1, top-5, and top-10 accuracies, and Mean Reciprocal Rank (MRR) of 0.52, 0.79, 0.85, and 0.64 for the four projects, which improves REVFinder by 61%, 23%, 8%, and 37%, respectively. The individual models of TIE also outperform REVFinder for almost all of the datasets and evaluation measures.

The main contributions of this paper are:

- We propose a hybrid and incremental approach TIE, which integrates both text mining and a file location-based approach, to recommend code-reviewers.
- We experiment on a broad range of datasets containing a total of 42,045 reviews to demonstrate the effectiveness of TIE. We show that TIE outperforms REVFinder by a substantial margin.

The remainder of the paper is organized as follows. Section II presents the motivation of our approach TIE. Section III describes an overview of TIE’s architecture. Section IV elaborates on the details of TIE. Section V presents the results of our comparative evaluation of TIE. Section VI surveys the related work. Finally, Section VII concludes the paper.

II. MOTIVATION

A typical review contains a number of fields including its upload time, description, file paths, and reviewers. The upload time field provides the time when a review request was submitted. The description field contains textual information explaining the change to be reviewed. The file path field lists the locations of files that are modified in the change. The reviewer field specifies a set of developers who are assigned to the review.

Figure 1 presents an example code review taken from the Gerrit’s system of the QT project². A developer Stephen Kelly made a code change and submitted it for review on May 2, 2012. The change avoided “the macro re-definition warning for QT_NO_EXCEPTIONS”, and modified the file



Change-Id:	I0c4b2d00dd567af17f22b733b93032ff1056fcbd		
Owner	Stephen Kelly (Unused account)		
Project	qt/qtbase		
Branch	master		
Topic			
Uploaded	May 2, 2012 7:16 PM	Time	
Updated	May 2, 2012 11:19 PM		
Status	Merged		
Commit Message			
Textual Content		Permalink 	
Avoid macro re-definition warning for QT_NO_EXCEPTIONS			
Change-Id: I0c4b2d00dd567af17f22b733b93032ff1056fcbd			
Reviewer			
Code-Review		Sanity-Review	
Stephen Kelly (Unused account)			
Qt Sanity Bot		✓	
Thiago Macieira		Reviewer	✓
File Path			
Commit Message			
M	src/corelib/global/qcompilerdetection.h		File Path

Fig. 1. Review 25000 from the Gerrit system of the QT project.

Upload Time: May 2, 2012 7:16 PM
Textual Content: Avoid undefined macro warning for `_GXX_EXPERIMENTAL_CXX0X_`.
The GCC use of this is already correct.
File Path: src/corelib/global/qcompilerdetection.h
Reviewers: Thiago Macieira

Fig. 2. Review 25001 from the Gerrit system of the QT project.

Upload Time: May 11, 2012 6:18 PM
Textual Content: clang: Use `__has_feature()` to detect C++11 features
Apple’s clang version is often reported as the next official clang version, but Apple clang does not support all the features that the final official clang does. Instead of using version based feature detection, use the `__has_feature()` built-in instead, so that we correctly `#define` the various `Q_COMPILER_*` macros.
File Path: src/corelib/global/qcompilerdetection.h
Reviewers: Thiago Macieira

Fig. 3. Review 25002 from the Gerrit system of the QT project.

Upload Time: May 19, 2012 12:08 AM
Textual Content: Remove STL from qfeatures.txt
QT_NO_STL is now no longer available
File Path: src/corelib/global/qfeatures.txt
Reviewers: Thiago Macieira

Fig. 4. Review 26572 from the Gerrit system of the QT project.

“src/corelib/global/qcompilerdetection.h”. Thiago Macieira reviewed the change, and approved it. Finally, Qt Sanity Bot verified the change, and integrated it into the main branch.

Figures 2, 3, and 4 present reviews 25001³, 25992⁴, and 26572⁵ in the Gerrit’s system of the QT project, respectively. Due to space limitation, we only show the values of some fields in the reviews. The change reviewed in review 25001 is to avoid the undefined macro warning for “`_GXX_EXPERIMENTAL_CXX0X_`”, the change in review 25002 is to correctly define various `Q_COMPILER_*` macros by using the “`__has_feature()`” function, and the change in

²<https://codereview.qt-project.org/#/c/25000>

³<https://codereview.qt-project.org/#/c/25001>

⁴<https://codereview.qt-project.org/#/c/25992>

⁵<https://codereview.qt-project.org/#/c/26572>

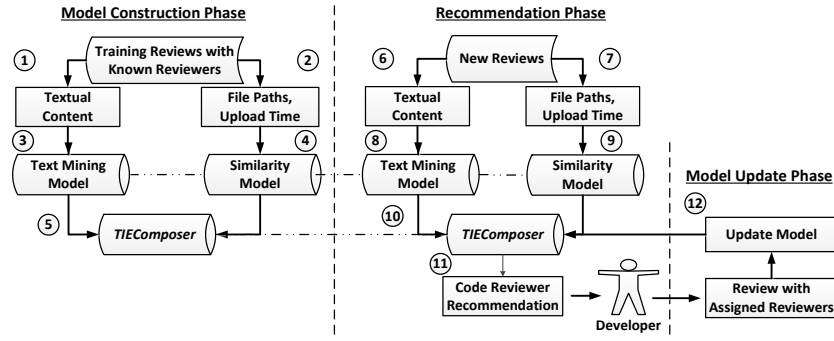


Fig. 5. Overall architecture of TIE.

review 26572 is to remove STL from “qfeatures.txt”. We notice that all of these changes were reviewed by the same person, Thiago Macieira.

Observations and Implications. From the above four reviews, we make the following observations:

- 1) The textual contents in the description field are good indicators to recommend appropriate reviewers. For example, the textual contents of the description field of reviews in Figures 1, 2, and 3 specify that the changes are about macros – the changes in review 25000 and 25001 are to avoid re-defined and undefined macro warnings, while the change in review 25992 is to correctly define the various `Q_COMPILER_*` macros.
- 2) The file paths could also help to recommend suitable reviewers. The same reviewer is likely to review multiple changes which modify the same files or files located in similar locations. For example, changes described in the reviews shown in Figures 1, 2, and 3 modify the same file “src/corelib/global/qcompilerdetection.h”. Also, the change described in the review shown in the Figure 4 modifies the file “src/corelib/global/qfeatures.txt” which is located in a similar location as files modified in the reviews shown in Figures 1, 2, and 3. All of these 4 reviews were assigned to the same person (Thiago Macieira).
- 3) A reviewer who reviews a code change at a particular time point is likely to review other changes in the near future. We refer to this property as the *temporal locality* of the reviewers. For example, all of the above four changes are submitted for reviews within a 17-day interval (May 2-19 2012), and Thiago Macieira reviewed all of them.

The above observations tell us that textual contents and file paths provide some useful and yet hidden information that can help us recommend reviewers. Text mining techniques that mine hidden information from a large text corpus, could potentially be used to solve the recommendation problem. Similarly, a file location-based approach which measures the similarity of the changed file paths with file paths of files reviewed in the past, could also be used to solve the problem. The last observation informs us that to improve efficiency, we can leverage the temporal localness property and only analyze the most recent code reviews.

In the following section, we describe the details of TIE, which is based on the above observations; it combines an incremental text mining technique and a time-aware file location-based approach to recommend reviewers. The file location-based approach potentially needs to compare a large number of file paths (especially for projects with a long history of code reviews) and thus we make it time-aware to improve its efficiency while not reducing its effectiveness in recommending correct reviewers.

III. TIE ARCHITECTURE

Figure 5 presents the overall architecture of TIE, which contains three phases: model construction, recommendation, and model update. In the model construction phase, a composite model TIECOMPOSER is built from historical reviews with known reviewers. In the recommendation phase, the model is used to recommend a set of reviewers for a new review request. In the model update phase, the model is updated by using additional reviews with known reviewers. To simulate real-life usage of our tool, we allow the model to be updated. In practice, new review requests would be submitted and assigned to reviewers periodically; these new reviews can be used to update the model.

In the model construction phase, TIE first collects various information from a set of training reviews with known reviewers, i.e., the textual contents in the description field (Step 1) and the file paths and upload time (Step 2). For the textual contents in the reviews, TIE tokenizes them, removes stop words, stems each token (i.e., reduces it to its root forms, e.g., “wrote” and “written” are reduced to “writ”), and represents the stemmed non-stop-word tokens as a bag of words, c.f., [29], [32], which is then converted into a vector of weights following the vector space model [9]. Next, TIE builds a text mining model based on the processed text by leveraging a text classification technique (Step 3). The intuition of the text mining model is that the same reviewers are likely to review changes containing similar terms (words). TIE also uses a time-aware file location-based approach which builds a similarity model which measures similarities of the new and historical reviews (Step 4). The similarity is computed by measuring the similarities of changed file paths (i.e., paths of files modified in the new review request) and reviewed file paths (i.e., paths of files reviewed in historical reviews). The intuition of the file location-based approach is that the same

reviewers are likely to review changes to the same files or files in similar locations. Next, these two models are blended to construct the TIECOMPOSER model (Step 5).

In the recommendation phase, TIE is used to recommend a ranked list of reviewers to a new unassigned review request. TIE first extracts the change description, file paths, and upload time from the review (Steps 6 and 7). Then, it processes the textual contents in the description and inputs them into the text mining model built in the model construction phase (Step 8). It also inputs the file paths and upload time into the similarity model built in the model construction phase (Step 9). These two models will output two lists of reviewers, and these two lists are combined by leveraging the TIECOMPOSER model constructed in the model construction phase (Step 10). In practice, a developer will check the list of potential reviewers, and eventually assign the new review request to a set of reviewers. In the model update phase, we update TIE by using newly assigned reviews (Step 12).

IV. OUR PROPOSED APPROACH

In this section, we present the details of the main components of TIE; the text mining model, the similarity model, and our method to blend the text mining and similarity models.

A. Text Mining Model

The intuition of the text mining model is that similar changes are often described in a similar way, and a developer often reviews similar changes. To build a text mining model that can predict the best reviewers to assign to a review request, we make use of a text classification technique. Since TIE contains a model update phase, we should select a text classification technique which is incremental and produces an updatable model. In this paper, we leverage naive Bayes multinomial [17] to build the text mining model, as it is a fast and effective algorithm for text classification [17], and can be used as an incremental learning algorithm. For many other text mining algorithms, e.g., decision tree and SVM, they take a long time to be run on a raw text dataset which has a large number of features (every processed word is a feature), and the models built using these algorithms can not be updated with new reviews (the models have to be re-constructed with new reviews).

1) *Model Construction Phase*: Let us consider that there exists l potential reviewers in a project and denote the j^{th} reviewer as D^j . TIE first duplicates the training dataset into l binary datasets, one for each reviewer. Let us denote the i^{th} review in the j^{th} dataset as Rev_i^j . Each review consists of two parts: the textual contents in the review ($Text_i$), and a binary value R_i^j that indicates whether D^j reviews it ($R_i^j = 1$) or not ($R_i^j = 0$). Following vector space modeling [9], we represent the text in the i^{th} review as a vector of term weights denoted by $Text_i = \langle w_1, w_2, \dots, w_v \rangle$. The weight w_j denotes the number of times the j^{th} term t_j appears in the textual content of the i^{th} review.

Next, we construct l prediction models based on the l binary datasets by leveraging the naive Bayes multinomial

algorithm. The i^{th} model consists of a prior probability that D^i reviews a change (denoted as $P(D^i = 1)$) and a set of conditional probabilities that term t_j occurs in the reviews which are reviewed by D^i (denoted as $P(t_j|D^i = 1)$). The prior probability $P(D^i = 1)$ is computed as the ratio of the number of reviews reviewed by D^i and the total number of reviews. The conditional probability $P(t_j|D^i = 1)$ is computed as:

$$P(t_j|D^i = 1) = \frac{T_{D^i=1,t_j}}{\sum_{t \in v} T_{D^i=1,t}} \quad (1)$$

where $T_{D^i=1,t_j}$ denotes the number of times term t_j appears in the reviews which are reviewed by D^i .

2) *Recommendation Phase*: In the recommendation phase, for a new review R_{new} , we first input it into the l prediction models, and each prediction model P_i would output a confidence score $Conf_{Text}(R_{new}, D^i)$ that indicates the likelihood that D^i reviews it. The confidence score $Conf_{Text}(R_{new}, D^i)$ is computed as:

$$Conf_{Text}(R_{new}, D^i) = P(D^i = 1) \times \prod_{j=0}^v P(t_j|D^i = 1) \quad (2)$$

In the above equation, $P(D^i = 1)$ represents the prior probability that D^i reviews a change, $P(t_j|D^i = 1)$ represents the conditional probability that term t_j occurs in the reviews which are reviewed by D^i . Next, we rank reviewers based on these confidence scores, and output the reviewers with the highest scores.

3) *Model Update Phase*: For a newly assigned review, we could use them to update our text mining model. For each reviewer D^i and each term t_j in the newly assigned review, we update the prior probability $P(D^i = 1)$ and the conditional probability $P(t_j|D^i = 1)$.

B. Similarity Model

The intuition of the similarity model is that the same reviewers are likely to review changes to the same files or files in similar locations. Given a new unassigned change, our similarity model compares the file paths in the change with the file paths in the previous reviewed changes to recommend reviewers. Thus, our similarity model is a *lazy* model [13], i.e., we do not need to train an explicit model.

1) *Model Construction Phase*: In the model construction phase, for each review R_i , our similarity model stores its upload time, file paths, and reviewers. We denote the upload time of R_i as $Time_i$, its file paths as $FilePaths_i = \{Path_1, Path_2, \dots, Path_{n_i}\}$ where n_i represents the number of file paths changed in R_i , and its reviewers as $D_i = \{D_1, D_2, \dots, D_{r_i}\}$ where r_i represents the number of reviewers who are assigned to R_i .

2) *Recommendation Phase*: Given a new review R_{new} , let us denote its upload time as $Time_{new}$, and its file paths as $FilePaths_{new}$. In this phase, we will first compare the new review with existing reviews that have been gathered in the model construction phase. To leverage the temporal locality of reviewers, and reduce the amount of comparisons needed,

TIE only compares the new review request with the previous assigned reviews whose upload time is within the past M days before $Time_{new}$. By default, we set $M=100$. We denote the set of historical reviews to compare R_{new} with as $HSet$.

Algorithm 1 presents the detailed steps to compute the similarity score between R_{new} and a past assigned review R_i in $HSet$. This similarity score is denoted as $RevSim(R_{new}, R_i)$. Our approach computes the similarity score of the two reviews by comparing their file paths (i.e., the paths of files that are modified in the change that is to be/was reviewed). Let us denote the sets of file paths for R_{new} and R_i as $FilePaths_{new}$ and $FilePaths_i$, respectively. The similarity between these sets of paths is the average of the similarities of each possible pairings of paths, one from each set.

Algorithm 1 Compute similarity between two reviews.

```

1: RevSim( $R_{new}, R_i$ )
2: Input:
3:  $R_{new}$ : The new unassigned review.
4:  $R_i$ : A past assigned review.
5: Output: Similarity score of the two input reviews.
6: Method:
7:  $SumScore = 0$ ;
8: for all  $Path_n \in FilePaths_{new}$ , and  $Path_i \in FilePaths_i$  do
9:    $DFNames_n$  = Set of directory and file names in  $Path_n$ ;
10:   $DFNames_i$  = Set of directory and file names in  $Path_i$ ;
11:   $Common = DFNames_n \cap DFNames_i$ ;
12:   $SumScore += \frac{|Common|}{\max(|DFNames_n|, |DFNames_i|)}$ ;
13: end for
14: Return  $\frac{SumScore}{|FilePaths_{new}| \times |FilePaths_i|}$ .
```

To compute the similarity between a path $Path_n \in FilePaths_{new}$, and a path $Path_i \in FilePaths_i$, our approach first computes, for each path, a set of directory and file names in it (Lines 9–10). For example, let $P1 = \text{"src/corelib/global/qcompilerdetection.h"};$ its set of directory and file names is $\{\text{"src"}, \text{"corelib"}, \text{"global"}, \text{"qcompilerdetection.h"}\}$. Also let $P2 = \text{"src/corelib/local/compiler/qcompilerdetection.h"};$ its set of directory and file names is $\{\text{"src"}, \text{"corelib"}, \text{"local"}, \text{"compiler"}, \text{"qcompilerdetection.h"}\}$. Next, our approach computes the number of common directories between $Path_n$ and $Path_i$ (Line 11). For example, the common directory and file names between $P1$ and $P2$ are $\text{"src"}, \text{"corelib"},$ and $\text{"qcompilerdetection.h"}.$ Then, the similarity score between $Path_n$ and $Path_i$ is computed as the ratio of the number of common directory and file names and the maximum number of directory and file names in $Path_n$ and $Path_i$ (Line 12). For example, the similarity score for path $P1$ and $P2$ is $\frac{3}{\max(4,5)} = 0.6$.

We will sum up the similarity scores of all possible pairs of paths from $FilePaths_{new}$ and $FilePaths_i$, and finally compute the average similarity score by dividing the sum with the number of possible pairs (Line 14). The average similarity score of all possible pairings of paths is the similarity score of the two reviews: R_{new} and R_i .

After we have computed the similarity scores between R_{new} and each review in the set of historical reviews $HSet$, for

each reviewer D^i , we compute its confidence score, denoted as $Conf_{Path}(R_{new}, D^i)$, as follows:

$$Conf_{Path}(R_{new}, D^i) = \sum_{R_j \in HSet} RevSim(R_{new}, R_j) \times Exist(D^i, R_j) \quad (3)$$

In the equation above, $Exist(D^i, R_j) = 1$ if D^i reviewed R_j , and $Exist(D^i, R_j) = 0$ otherwise.

3) *Model Update Phase:* Similar to the model construction phase, in the model update phase, our approach simply adds up newly assigned reviews to our repository of historical reviews and stores their upload time, file paths, and reviewers.

C. TIECOMPOSER: A Composite Approach

In TIE, we have two prediction models: text mining model and similarity model. Each of the prediction model would recommend a ranked list of reviewers. In this section, we propose TIECOMPOSER, which combines these two prediction models by assigning weights to them.

Given a new review R_{new} , for each reviewer D^i , the text mining model and similarity model would output the confidence scores $Conf_{Text}(R_{new}, D^i)$ and $Conf_{Path}(R_{new}, D^i)$. TIECOMPOSER first normalizes these two scores to compute $NConf_{Text}(R_{new}, D^i)$ and $NConf_{Path}(R_{new}, D^i)$ as follows:

$$NConf_{Text}(R_{new}, D^i) = \frac{Conf_{Text}(R_{new}, D^i)}{\sum_{D^j \in AllRev} Conf_{Text}(R_{new}, D^j)}$$

$$NConf_{Path}(R_{new}, D^i) = \frac{Conf_{Path}(R_{new}, D^i)}{\sum_{D^j \in AllRev} Conf_{Path}(R_{new}, D^j)} \quad (4)$$

In the above equations, $AllRev$ refers to the set of all reviewers in a software project. Next, TIECOMPOSER combines the normalized scores for each reviewer D^i as follows:

$$Composer(R_{new}, D^i) = \alpha \times NConf_{Text}(R_{new}, D^i) + (1 - \alpha) \times NConf_{Path}(R_{new}, D^i) \quad (5)$$

In the above equation, $\alpha \in [0, 1]$. If $\alpha = 0$, the TIECOMPOSER model is the same as the similarity model, and if $\alpha = 1$, the TIECOMPOSER model is the same as the text mining model. The value of α can be empirically determined. By default, we set the value of α as 0.3.

Given a new review R_{new} , for each reviewer D_i , we compute its TIECOMPOSER score (i.e., $Composer(R_{new}, D^i)$). Then, we rank the reviewers according to their TIECOMPOSER scores, and finally output the reviewers with the highest scores.

V. EXPERIMENTS AND RESULTS

In this section, we evaluate the performance of TIE. The experimental environment is a 64-bit, Intel Xeon 2.00GHz server with 80GB RAM running Windows Server 2008.

A. Experiment Setup

We use the datasets provided by Thongtanunam et al. [28] which contain a total of 42,045 reviews. The status of each of these reviews is either “merged” or “abandoned”, and each of them has at least one file path (i.e., at least one file is changed and reviewed) [28]. Since the datasets collected

TABLE I
STATISTICS OF THE COLLECTED DATA.

Projects	Time Period	# Revi.	# Re.	# Files	# Avg.Re.
Android	2008-10 – 2012-01	5,126	94	26,840	1.06
OpenStack	2011-07 – 2012-05	6,586	82	16,953	1.44
QT	2011-05 – 2012-05	23,810	202	78,401	1.07
LibreOffice	2012-03 – 2014-06	6,523	63	35,273	1.01

by Thongtanunam et al. do not contain any textual content information, we also crawl the projects’ respective Gerrit systems to extract the contents of the description field of each of the reviews in the datasets. We use WVTool [30] to remove stop words, do stemming, and produce vector space models from the textual contents of the reviews. We remove terms (words) which appear only once since these terms are likely not to have significantly contributions for code-reviewer recommendation.

Table I presents the statistics of the collected data. The columns correspond to the project name (Project), the time period of collected reviews (Time Period), the number of collected reviews (# Revi.), the number of unique reviewers (# Re.), the number of unique modified files (# Files), and the average number of code-reviewers per review (# Avg. Re.).

To simulate the usage of our approach in practice, we use the same longitudinal data setup used by Thongtanunam et al. [28]. The reviews extracted from each review repository in Table I are first sorted in chronological order of their upload time. The process proceeds as follows: first, we train a TIE model by using the first review, and test the trained model by using the second review, then we update the TIE model by using the second review (with its ground truth reviewers). Next, we test using the third review, and update the TIE model by using the third review, and so on. We then compute the average accuracy which is defined as the ratio between the total number of correct predictions and the total number of predictions (i.e., the total number of reviews minus one).

The default parameters of TIE are as follows: we set the number of past days in the similarity model M as 100, and the α value in the TIECOMPOSER component as 0.3. We compare our approach with REVFINDER proposed by Thongtanunam et al. [28]. Since we use the same datasets and the same experiment setup as Thongtanunam et al.’s, we use the results reported in their paper.

B. Evaluation Metrics

To evaluate TIE, we use the top-k prediction accuracy and the Mean Reciprocal Rank (MRR) [9], which are the same evaluation metrics used to evaluate REVFINDER, and are commonly used in evaluating recommendation systems in the software engineering literature [21], [27], [33], [36].

1) *Top-k Prediction Accuracy*: Top-k prediction accuracy is the percentage of reviews where their ground truth code-reviewers are ranked in the top-k positions in the returned ranked lists of reviewers. Given a review r , if at least one of its top-k code-reviewers actually reviews r , we consider the reviewers are correctly recommended, and set the value $isRecomm(r, top-k)$ to 1; else we consider the reviewers are

wrongly recommended, and set the value $isRecomm(r, top-k)$ to 0. Given a set of reviews $Reviews$, the top-k prediction accuracy $Top@k$ is computed as:

$$Top@k = \frac{\sum_{r \in Reviews} isRecomm(r, top-k)}{|Reviews|} \quad (6)$$

The higher the metric value, the better a code-reviewer recommendation technique performs. In this paper, we set $k = 1, 3, 5$, and 10 .

2) *Mean Reciprocal Rank (MRR)*: MRR is a popular metric used to evaluate an information retrieval technique [9]. Given a query (in our case: a review r), its reciprocal rank is the multiplicative inverse of the rank of the first correct document (in our case: code-reviewer) in a rank list produced by a ranking technique (in our case: a code reviewer recommendation technique). Mean Reciprocal Rank (MRR) is the average of the reciprocal ranks of reviews in a set of reviews. The MRR of a set of reviews $Reviews$ is computed as:

$$MRR(R) = \frac{1}{|Reviews|} \sum_{r \in Reviews} \frac{1}{rank(r)} \quad (7)$$

In the above equation, $rank(r)$ refers to the rank of the first correctly recommended code-reviewer in the ranked list returned by a code-reviewer recommendation technique for review r .

C. Research Questions

RQ1: How effective is TIE in recommending code-reviewers? How much improvement can it achieve over the state-of-the-art approach?

Motivation. The more accurate TIE is, the more benefit TIE would give to its users. Thus, in this research question, we evaluate the effectiveness of TIE and compare it with the state-of-the-art approach.

Approach. To answer RQ1, we compare TIE with REVFINDER proposed by Thongtanunam et al. [28]. We evaluate them by using the longitudinal data setup, and record the top-k prediction accuracies ($k = 1, 3, 5$, and 10), and MRR.

Results. Tables II presents the top-k prediction accuracies ($k = 1, 3, 5$, and 10) and MRR values of TIE and REVFINDER. We notice that TIE outperforms REVFINDER by a substantial margin. On average across the 4 projects, TIE achieves top-1, top-3, top-5, and top-10 prediction accuracies, and MRR values of 0.52, 0.73, 0.79, 0.85, and 0.64, which outperform REVFINDER results by 61%, 33%, 23%, 8%, and 37%, respectively.

For LibreOffice, we notice TIE achieves the most improvement over REVFINDER; it improves REVFINDER by 217%, 94%, 58%, 30%, and 110% in terms of top-1, top-3, top-5, and top-10 prediction accuracies, and MRR respectively. Upon closer inspection, we find that the textual contents in the description field of LibreOffice’s reviews provide useful information to recommend suitable reviewers. These textual contents are ignored by REVFINDER.

For QT, we notice that REVFINDER achieves better performance than TIE in terms of top-10 prediction accuracy. Upon

TABLE II
TOP-K PREDICTION ACCURACIES (K = 1, 3, 5, AND 10) AND MRR OF TIE COMPARED WITH REVFinder (REV.).

Project	Top-1			Top-3			Top-5			Top-10			MRR		
	TIE	Rev.	%Imp	TIE	Rev.	%Imp	TIE	Rev.	%Imp	TIE	Rev.	%Imp	TIE	Rev.	%Imp
Android	0.57	0.46	24%	0.81	0.71	14%	0.87	0.79	10%	0.92	0.86	7%	0.70	0.60	17%
OpenStack	0.43	0.38	13%	0.73	0.66	11%	0.83	0.77	8%	0.91	0.87	5%	0.60	0.55	9%
QT	0.30	0.20	50%	0.45	0.34	32%	0.52	0.41	27%	0.62	0.69	-10%	0.41	0.31	32%
LibreOffice	0.76	0.24	217%	0.91	0.47	94%	0.93	0.59	58%	0.96	0.74	30%	0.84	0.40	110%
Average.	0.52	0.32	61%	0.73	0.55	33%	0.79	0.64	23%	0.85	0.79	8%	0.64	0.47	37%

TABLE III
TOP-K PREDICTION ACCURACIES (K = 1, 3, 5, AND 10) AND MRR VALUES OF TIE COMPARED WITH THE TEXT MINING MODEL (TEXT.) AND THE SIMILARITY MODEL (SIM.). THE BEST TOP-K ACCURACIES AND MRR VALUES ARE IN BOLD.

Project	Top-1			Top-3			Top-5			Top-10			MRR		
	TIE	Text.	Sim.	TIE	Text.	Sim.	TIE	Text.	Sim.	TIE	Text.	Sim.	TIE	Text.	Sim.
Android	0.57	0.48	0.51	0.81	0.70	0.77	0.87	0.78	0.84	0.92	0.86	0.91	0.70	0.61	0.65
OpenStack	0.43	0.27	0.48	0.73	0.50	0.75	0.83	0.61	0.84	0.91	0.76	0.92	0.60	0.42	0.64
QT	0.30	0.28	0.17	0.45	0.45	0.32	0.52	0.53	0.40	0.62	0.64	0.54	0.41	0.40	0.29
LibreOffice	0.76	0.88	0.31	0.91	0.94	0.53	0.93	0.95	0.64	0.96	0.97	0.78	0.84	0.91	0.46
Average.	0.52	0.48	0.37	0.73	0.65	0.59	0.79	0.72	0.68	0.85	0.81	0.78	0.64	0.59	0.51

closer inspection, different from the other datasets, for QT, we find that the top-10 lists recommended by the text mining and similarity models of TIE are substantially different (they contain only a few common candidate reviewers), and thus when we combine these two lists, the top-10 accuracy of TIE is not optimal. However, considering that the average number of code-reviewers per review of QT is 1.07, in practice, top-1, top-3, and top-5 accuracies would be more reasonable metrics to evaluate the performance of TIE and REVFinder.

RQ2: What is the performance of the text mining model and similarity model in TIE? And what is the benefit of our TIECOMPOSER?

Motivation. TIE has two prediction models (text mining model and similarity model). In this RQ, we would like to investigate the performance of each of them. We want to see whether the combination of the two prediction models can achieve better result than the individual prediction models.

Approach. To answer RQ2, we compare TIE with its two prediction models: text mining model and similarity model. We evaluate these approaches by using the longitudinal data setup, and record the top-k prediction accuracies (k = 1, 3, 5, and 10), and MRR.

Results. Table III presents the top-k prediction accuracies (k = 1, 3, 5, and 10) and MRR values of TIE, the text mining model and the similarity model. On average across the 4 projects, TIE improves the text mining model by 8%, 12%, 10%, 5%, and 8%, and the similarity model by 41%, 24%, 16%, 9%, and 25% in terms of top-1, top-3, top-5, and top-10 prediction accuracies, and MRR, respectively. Thus, the combination of text mining model and similarity model helps to improve the effectiveness of the individual models.

From Table III, we notice for OpenStack, TIECOMPOSER performs better than the text mining model, and worse than the similarity model. And for LibreOffice, TIECOMPOSER performs better than the similarity model, and worse than the text mining model. In practice, since we do not know which prediction model would perform better, and the combination of these two models (i.e., TIECOMPOSER) would typically at least perform better than one of the prediction models, it is best to use TIECOMPOSER. Also, the weight α can be empirically

tuned. For OpenStack, we can set α to 0, while for LibreOffice, we can set α to 1. Thus, we recommend developers to use TIECOMPOSER instead of one of the two prediction models in practice.

To gain more insight why the text mining model is more accurate than the similarity model (or vice versa) for some datasets, we manually read some reviews from the four datasets. For QT and LibreOffice, terms in description fields of reviews are often good indicators to find suitable reviewers. For example, a LibreOffice developer “Fridrich” always reviews changes that contain terms “graph” and “jpeg”. Similarity, for Android and OpenStack, file paths are often good indicators. For example, an Android developer “David Turner” reviews 295 changes to files in path “tools/emulator/opengl/host/libs/Translator”.

Moreover, we notice that in general the performance of the text mining model and the similarity model (used alone) are better than REVFinder. On average across the 4 projects, the text mining model outperforms REVFinder by 30%, 10%, 6%, 4%, and 16% in terms of the top-1, top-3, top-5, and top-10 prediction accuracies, and MRR, and the similarity model outperforms REVFinder by 16%, 7%, 6%, -1%, and 9%, respectively.

RQ3: How does the performance of TIE vary for various settings of parameter α ?

Motivation. By default, we set the α value in Equation 5 as 0.3. α is used to assign the weights of the text mining model and the similarity model. We would like to investigate the performance of TIE for various α values.

Approach. To address RQ3, we increase α values from 0 to 1 with an interval of 0.1. We evaluate TIE with different α values by using the longitudinal data setup, and record the top-k prediction accuracies (k = 1, 3, 5, and 10), and MRR.

Results. Figure 6 presents the top-k prediction accuracies (k = 1, 3, 5, and 10) and MRR values of TIE for various α values for Android, OpenStack, QT, and LibreOffice, respectively. For Android, we notice that TIE’s top-k prediction accuracies and MRR values slightly increase when we increase α from 0 to 0.3, and slightly decrease when we increase α from

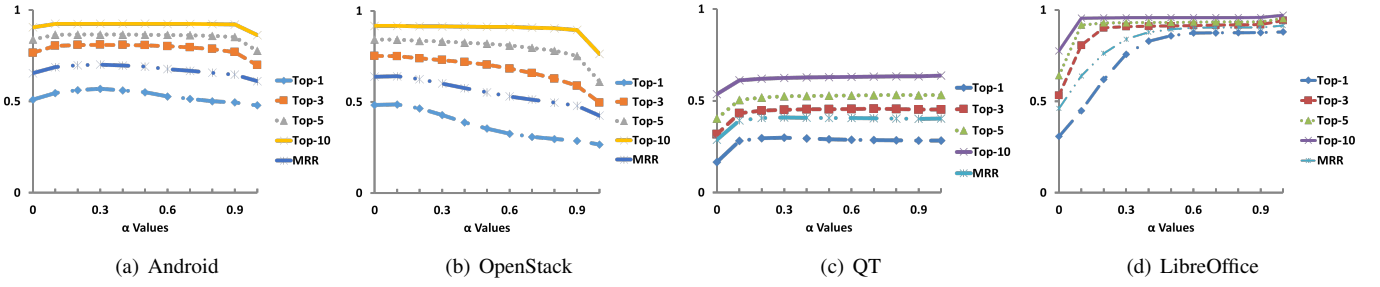


Fig. 6. Top-k prediction accuracies ($k = 1, 3, 5$, and 10) and MRR values of TIE for various α values.

0.3 to 1. For OpenStack, its top-k prediction accuracies and MRR values slightly decrease when we increase α from 0 to 0.9, and they decrease quickly when we increase α from 0.9 to 1. For QT, its top-k prediction accuracies and MRR values increase quickly when we increase α from 0 to 0.1, and slightly increase when we increase α from 0.1 to 1. For LibreOffice, its top-k prediction accuracies and MRR values increase quickly when we increase α from 0 to 0.4, and slightly increase when we increase α from 0.4 to 1.

This is consistent with our findings to answer RQ2: in QT and LibreOffice, the text mining model shows better performance than the similarity model, thus the increase of α will increase the performance of TIE. On the other side, in Android and OpenStack, the similarity model shows better performance than the text mining model, thus the increase of α will decrease the performance of TIE.

RQ4: How does the effectiveness and efficiency of TIE vary for various settings of the number of past days M in the similarity model? And what is the benefit of considering the reviews in the past M days?

Motivation. By default, we set the number of past days M in the similarity model as 100. We would like to investigate the effect of varying the number of past days M . We are also interested to investigate whether it is beneficial to consider only reviews in the past M days (i.e., whether the improvement in efficiency comes with a large reduction in effectiveness) or to consider all previous reviews.

Approach. To answer RQ4, we set M to 50, 100, 150, 200, 250, and denote TIE with each setting as TIE(50), TIE(100), TIE(150), TIE(200), and TIE(250). We also evaluate the effectiveness and efficiency of TIE without considering the number of past days and denote it as TIE(ALL), i.e., we consider all previous reviews. We evaluate TIE with different M values by using the longitudinal data setup, and record the top-k prediction accuracies ($k = 1, 3, 5$, and 10), MRR and recommendation time. Note that the model construction time of all of the above variants are the same.

Results. Figure 7 presents the top-k prediction accuracies ($k = 1, 3, 5$, and 10) and MRR values of TIE for various M values for the Android, OpenStack, QT, and LibreOffice datasets, respectively. We notice that in general, the performance of TIE is stable for various M values. For example, for Android, the top-5 prediction accuracies, and MRR values are varied only between 0.86 – 0.87, and 0.69 – 0.70. For LibreOffice, except

TABLE IV
RECOMMENDATION TIME PER REVIEW OF TIE(100) COMPARED WITH TIE(ALL) (IN SECONDS).

Project	TIE(100)	TIE(ALL)
Android	0.21	0.42
OpenStack	0.22	0.44
QT	0.25	0.45
LibreOffice	0.28	0.48
Average.	0.24	0.45

for top-1 prediction accuracy and MRR, the performance of TIE remains stable as M is increased.

Table IV presents the recommendation time per review of TIE(100), the default setting of TIE, compared with that of TIE(ALL) (in seconds). On average across the 4 projects, to recommend a list of reviewers to a change, TIE(100) needs 0.24 seconds while TIE(ALL) needs 0.45 seconds.

D. Threats to Validity

Threats to internal validity relates to errors in our code and experiment bias. We have double-checked our code, still there could be errors that we did not notice. Also, in this paper, we use a longitudinal data setup to simulate the actual usage of TIE. In practice, we can only use reviews submitted before to build a model, and update the model with newly assigned reviews. Threats to external validity relate to the generalizability of our results. We have analyzed 42,045 reviews from four open source projects. In the future, we plan to reduce this threat further by analyzing even more reviews from additional projects. Threats to construct validity refer to the suitability of our evaluation measures. We use top-k prediction accuracies and MRR which are also used by past studies to evaluate the effectiveness of a code-reviewer recommendation technique [28], and various automated software engineering techniques [21], [27], [33], [36]. Thus, we believe there is little threat to construct validity.

VI. RELATED WORK

Studies on Code Review. Thongtanunam et al. study code-reviewer recommendation problems in four open source projects [28]. They find that review requests with a reviewer assignment problem take on average an additional 12 days to get approved, and they propose REVFINDER, a file location-based approach to recommend code-reviewers. Our work extends their work in the following ways:

- 1) Our work considers more features than REVFINDER. Besides the file paths in the reviews, we also leverage

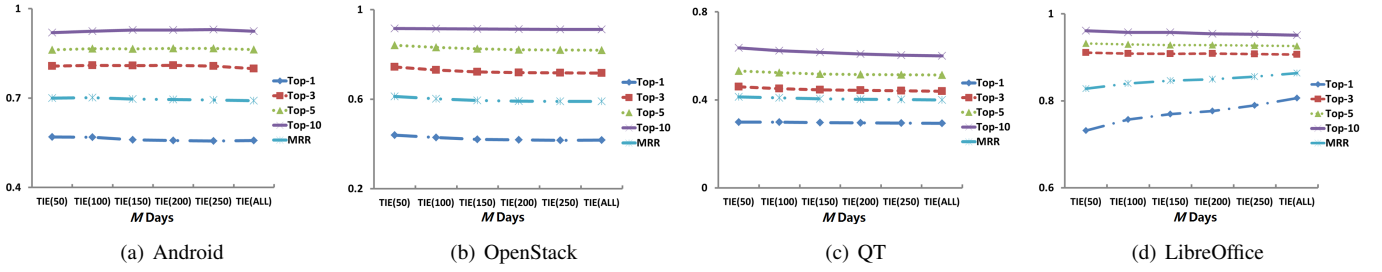


Fig. 7. Top-k prediction accuracies ($k = 1, 3, 5$, and 10) and MRR values of TIE for various past M days settings.

the textual contents in the description fields of reviews to recommend code-reviewers.

- 2) We propose a more accurate file location-based approach (our similarity model).
- 3) We propose a composite model which combines our text mining model and our similarity model to achieve a better performance.

There are a number of studies that investigate code review practice [8], [10], [12], [15], [18], [20], [22]. Rigby et al. perform an empirical study on the code review practice followed by an open source project; they analyze a repository of Apache server's code reviews and investigate the review process, frequency of reviews, level of participation, review interval, and quality of reviews [22]. Baysal et al. study the influence of non-technical factors (e.g., patch size, priority) on code reviews of the WebKit project [10]. Jiang et al. empirically investigate the relationship between accepted patches and reviewing time on Linux [15]. McIntosh et al. study the impact of code review coverage and code review participation on the quality of several software systems, e.g., QT, VTK, and ITK [18]. Morales et al. investigate the impact of code review practice on the quality of software design [20]. Bacchelli and Bird observe, interview, and survey practitioners in Microsoft and manually categorize review comments made by various teams in Microsoft to investigate the motivations, challenges, and outcomes of the code review process [8]. Gousios et al. analyze projects in GitHub and find that various factors such as developer contribution history, size of a project, and test coverage would affect code review time [12]. Our work is orthogonal to the above studies: we focus on code-reviewer recommendation, which is a different problem than the ones addressed in the above studies.

Studies on Developer Recommendation. There have been a number of studies on developer recommendation in software engineering literature. Mockus and Herbsleb propose a tool that uses data from change management systems to recommend developers with relevant expertise in geographically distributed software projects [19]. Surian et al. analyze a developer collaboration network extracted from SourceForge to recommend suitable developers to join project teams [26]. Anvik et al. propose the usage of machine learning techniques such as Naive Bayes, support vector machine, and decision tree to recommend fixers to bug reports of Eclipse, Firefox, and GCC projects [6]. Kagdi et al. propose an

approach to recommend developer for a change request by leveraging feature location techniques [16]. Shokripour et al. propose a two-phase approach to recommend fixers to bug reports [24]. Their approach first determines candidate bug locations, and then uses a simple term weighting scheme to recommend potential bug fixers. Xia et al. propose DevRec which integrates a bug report-based analysis and a developer-based analysis to recommend developers who could help to resolve bug reports [34], [35]. Hossen et al. propose iMacPro that integrates source code authors, maintainers, and change proneness to assign a developer to a change request [14].

Our approach addresses a different problem (code reviewer recommendation) than those considered in the above-mentioned past studies (expert developer recommendation, and bug fixer recommendation). Code reviews have a different set of characteristics than change history data and bug reports analyzed in these prior studies, e.g., in code review, we know exactly the files that are going to be reviewed. Our approach leverages the two characteristics of code reviews, file paths and short descriptions, to accurately recommend code-reviewers.

VII. CONCLUSION AND FUTURE WORK

In this paper, we propose a hybrid and incremental approach TIE which utilizes the advantages of text mining and a file location-based approach to recommend code-reviewers. TIE integrates an incremental text mining model which analyzes the textual contents in the description field of reviews, and a similarity model which measures the similarity of newly changed files' paths and past reviewed files' paths. To investigate the benefits of TIE, we perform a large-scale experiment on four open source projects containing a total of 42,045 reviews. The experimental results show that TIE can achieve average top-1, top-5, and top-10 accuracies, and MRR of 0.52, 0.79, 0.85, and 0.64 across the four projects, which outperforms the results achieved by the state-of-the-art approach REVFINDER, by a substantial margin.

In the future, we plan to evaluate TIE with more reviews from more software projects, and develop a better technique that can recommend code-reviewers with higher accuracy.

Acknowledgment. This research was partially supported by China Knowledge Centre for Engineering Sciences and Technology (No. CKCEST-2014-1-5), and National Key Technology R&D Program of the Ministry of Science and Technology of China (No. 2015BAH17F01).

REFERENCES

- [1] Android code review system. <https://android-review.googlesource.com/>.
- [2] Libreoffice code review system. <https://gerrit.libreoffice.org/>.
- [3] Openstack code review system. <https://review.openstack.org/>.
- [4] Qt code review system. <https://codereview.qt-project.org/>.
- [5] A. F. Ackerman, P. J. Fowler, and R. G. Ebenau. Software inspections and the industrial production of software. In *Proceedings of the symposium on Software validation: inspection-testing-verification-alternatives*, pages 13–40. Elsevier North-Holland, Inc., 1984.
- [6] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, pages 361–370. ACM, 2006.
- [7] A. Aurum, H. Petersson, and C. Wohlin. State-of-the-art: software inspections after 25 years. *Software Testing, Verification and Reliability*, 12(3):133–154, 2002.
- [8] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 712–721. IEEE Press, 2013.
- [9] R. Baeza-Yates, B. Ribeiro-Neto, et al. *Modern information retrieval*, volume 463. ACM press New York, 1999.
- [10] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey. The influence of non-technical factors on code review. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*, pages 122–131. IEEE, 2013.
- [11] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 38(2.3):258–287, 1976.
- [12] G. Gousios, M. Pinzger, and A. v. Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355. ACM, 2014.
- [13] J. Han, M. Kamber, and J. Pei. *Data mining, southeast asia edition: Concepts and techniques*. Morgan kaufmann, 2006.
- [14] M. K. Hossen, H. Kagdi, and D. Poshyanyk. Amalgamating source code authors, maintainers, and change proneness to triage change requests. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 130–141. ACM, 2014.
- [15] Y. Jiang, B. Adams, and D. M. German. Will my patch make it? and how fast?: case study on the linux kernel. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 101–110. IEEE Press, 2013.
- [16] H. Kagdi, M. Gethers, D. Poshyanyk, and M. Hammad. Assigning change requests to software developers. *Journal of Software: Evolution and Process*, 24(1):3–33, 2012.
- [17] A. McCallum, K. Nigam, et al. A comparison of event models for naive bayes text classification. In *AAAI-98 workshop on learning for text categorization*, volume 752, pages 41–48. Citeseer, 1998.
- [18] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vt, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 192–201. ACM, 2014.
- [19] A. Mockus and J. D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the 24th international conference on software engineering*, pages 503–512. ACM, 2002.
- [20] R. Morales, S. McIntosh, and F. Khomh. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 171–180. IEEE Press, 2015.
- [21] S. Rao and A. Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 43–52. ACM, 2011.
- [22] P. C. Rigby, D. M. German, and M.-A. Storey. Open source software peer review practices: a case study of the apache server. In *Proceedings of the 30th international conference on Software engineering*, pages 541–550. ACM, 2008.
- [23] P. C. Rigby and M.-A. Storey. Understanding broadcast based peer review on open source software projects. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 541–550. ACM, 2011.
- [24] 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 the 10th Working Conference on Mining Software Repositories*, pages 2–11. IEEE Press, 2013.
- [25] F. Shull and C. Seaman. Inspecting the history of inspections: An example of evidence-based technology diffusion. *Software, IEEE*, 25(1):88–90, 2008.
- [26] D. Surian, N. Liu, D. Lo, H. Tong, E. Lim, and C. Faloutsos. Recommending people in developers’ collaboration network. In *Proceedings of the 18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17-20, 2011*, pages 379–388, 2011.
- [27] 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*, pages 365–375. ACM, 2011.
- [28] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto. Who should review my code? In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 141–150. IEEE Press, 2015.
- [29] Y. Tian, C. Sun, and D. Lo. Improved duplicate bug report identification. In *16th European Conference on Software Maintenance and Reengineering, CSMR*, pages 385–390, 2012.
- [30] D. Tutorial. The word vector tool. <http://sourceforge.net/projects/wvtool/>
- [31] L. G. Votta Jr. Does every inspection need a meeting? *ACM SIGSOFT Software Engineering Notes*, 18(5):107–114, 1993.
- [32] S. Wang, D. Lo, Z. Xing, and L. Jiang. Concern localization using information retrieval: An empirical study on linux kernel. In *Proceedings of the 18th Working Conference on Reverse Engineering, WCRE*, pages 92–96, 2011.
- [33] X. Xia, D. Lo, X. Wang, C. Zhang, and X. Wang. Cross-language bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 275–278. ACM, 2014.
- [34] X. Xia, D. Lo, X. Wang, and B. Zhou. Accurate developer recommendation for bug resolution. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*, pages 72–81. IEEE, 2013.
- [35] X. Xia, D. Lo, X. Wang, and B. Zhou. Dual analysis for recommending developers to resolve bugs. *Journal of Software: Evolution and Process*, 2015.
- [36] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 14–24. IEEE, 2012.