Reviewer Recommendation using Software Artifact Traceability Graphs

Emre Sülün
emre.sulun@ug.bilkent.edu.tr
Bilkent University
Ankara, Turkey

Eray Tüzün
eraytuzun@cs.bilkent.edu.tr
Bilkent University
Ankara, Turkey

Uğur Doğrusöz
ugur@cs.bilkent.edu.tr
Bilkent University
Ankara, Turkey

ABSTRACT

Various types of artifacts (requirements, source code, test cases, documents, etc.) are produced throughout the lifecycle of a software. These artifacts are often related with each other via *traceability links* that are stored in modern application lifecycle management repositories. Throughout the lifecycle of a software, various types of changes can arise in any one of these artifacts. It is important to review such changes to minimize their potential negative impacts. To maximize benefits of the review process, the reviewer(s) should be chosen appropriately.

In this study, we reformulate the reviewer suggestion problem using software artifact traceability graphs. We introduce a novel approach, named RSTrace, to automatically recommend reviewers that are best suited based on their familiarity with a given artifact. The proposed approach, in theory, could be applied to all types of artifacts. For the purpose of this study, we focused on the source code artifact and conducted an experiment on finding the appropriate code reviewer(s). We initially tested RSTrace on an open source project and achieved top-3 recall of 0.85 with an MRR (mean reciprocal ranking) of 0.73. In a further empirical evaluation of 37 open source projects, we confirmed that the proposed reviewer recommendation approach yields promising top-k and MRR scores on the average compared to the existing reviewer recommendation approaches.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; **Collaboration in software development**.

KEYWORDS

suggesting reviewers, reviewer recommendation, code review, software traceability, pull-request review, modern code review

ACM Reference Format:
Emre Sülün, Eray Tüzün, and Uğur Doğrusöz. 2019. Reviewer Recommendation using Software Artifact Traceability Graphs. In *The Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE'19), September 18, 2019, Recife, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3345629.3345637

1 INTRODUCTION

Due to its nature, through the lifecycle of a software, many different kinds of changes occur to the software. These changes might have intended consequences such as successfully adding new functionality or modifying the existing functionality. However, in some cases, the changes have unintended consequences such as introducing a new bug while fixing another; thus, these changes need to be carefully reviewed. To maximize benefits of a review, the reviewer(s) should be chosen appropriately. However, choosing the right reviewer(s) might not be a trivial task, especially in large projects. Automatically suggesting appropriate reviewers have two main benefits: (1) reducing overall review time by automatically assigning reviewers [22] (2) increasing the review quality and thus reducing the potential errors related to the reviewed artifact.

To find the most appropriate reviewer for reviewing an artifact, we mine the information stored in software artifact traceability graphs. In a typical software project, various types of artifacts (requirements, source code, test cases, document, risks, etc.) are produced throughout the lifecycle. These artifacts are not independent from each other and are often interlinked with each other via so-called traceability links. These artifacts and their traceability link information are typically stored in modern application lifecycle management (ALM) tools [23]. With the increased popularity of DevOps and Agile methodologies, and explosion of ALM tools usage [9], it is relatively easier to access this traceability information.

Previously, we reported our initial experience with a reviewer recommendation algorithm in [21]. In this study, we extend our previous work and introduce RSTrace for reviewer suggestion using software artifact traceability graphs. We address two research questions in this study:

RQ1: How does the accuracy of RSTrace compare to state-of-the-art reviewer suggestion approaches?

RQ2: How does the performance of RSTrace vary according to data availability?

To show the applicability of our approach, we conducted an initial experiment on an open-source project, Qt 3D Studio. This experiment was used to calibrate our approach. We have further evaluated our approach on 37 more open source projects. We confirm that our proposed reviewer recommendation approach obtains promising average top-1, top-3, top-5 accuracy and MRR scores than existing reviewer recommendation approaches.

In the next section, we describe background information related to graph analysis and reviewer suggestion. In Section 3, we formulate the reviewer suggestion problem using software artifact traceability links and provide details related to our approach. The practical tool support for RSTrace that is integrated with GitHub is explained in Section 4. In Section 5, to calibrate our approach, we ran RSTrace in a public data set derived from an open source project (Qt 3D Studio). In Section 6, we compare our approach with other approaches for code reviewer recommendation problem and discuss the results. Finally, Section 7 presents our concluding remarks.

2 BACKGROUND AND RELATED WORK

In the following subsections, we provide background information related to use of graph analysis in different software engineering problems, and the reviewer suggestion problem.

2.1 Graph Analysis

Graphs are useful for analyzing connected and relational data. Graph analysis is often supported with interactive visualization in many fields from security to computer and social networks. For instance, Hegeman and Iosup systematically identify the applications of graph analysis in different industries [5]. Use case areas consist of many engineering disciplines and social sciences. Software engineering is no exception. Zimmermann and Nagappan use network measures like eigenvector and degree centrality to predict defects [31]. Zhang et al. apply social network analysis on developers and create a communication network between developers [30]. Similarly, Yang uses social network metrics to find the most important contributor in an open source software project [27] by using centrality metrics such as degree, betweenness and closeness. In this study, we define a new metric based on graph analysis to calculate the best possible reviewer for a given artifact.

2.2 Reviewer Suggestion

Reviewer recommendation is one of the application areas of expert recommendation in software engineering. Reviewer recommendation could be in the form of code reviewer assignment, bug assignment and contributor recommendation. Many different methods were proposed to determine which developers should be assigned to fix a bug [2] [1] [15]. Researchers also propose techniques to suggest expert developers to contribute to an open source project by analyzing their past projects and technology stack [18].

In this study we mainly focus on automated code reviewer assignment problem. Code review, a known best practice, is an important process in software development to improve software quality and reduce defects. Formal inspection of code change can be time consuming and labor intensive. Modern code review, on the other hand, is a lightweight approach to code review, which is usually informal and supported by tools. It is widely adopted by industrial companies [20] [14] and open source projects on GitHub.

Popular projects usually get lots of pull requests. Thus, manually assigning those requests to a core member for review can be time consuming. Thongtanunam et al. finds that when code-reviewers are assigned manually, approval process takes around 12 more days compared to automatic assignments [22]. Reviewer assignment may not be straightforward for all reviews, especially in large projects. In such large projects, many developers may be qualified for a review, but it is essential to find the most qualified one for a thorough review. This may be tedious and time consuming. Therefore, automatic reviewer assignment can potentially reduce workload of developers and overall time of software development.

In the literature, there are many studies that propose different approaches to find suitable reviewers for a given changeset. Lee et al. introduced a graph-based approach to find reviewers in OSS projects [11], where they create a graph that connects only developers and source code files. In this study, in addition to developer and source code files, we include all other types of software artifacts provided that the data exists. Balachandran creates a tool called Review Bot that analyzes the change history of lines [3]. Thongtanunam et al. proposes a reviewer recommendation technique based on file path similarity [22]. Jiang et al. use support vector machines to find an appropriate core member in OSS projects [7]. In another study, Jeong et al. propose a method to suggest reviewers based on Bayesian Networks [6]. Yu et al. propose an approach by combining comment networks with traditional approaches to find suitable developers for pull-requests in GitHub projects [28]. Cheng et al. propose a method called RevRec to recommend reviewers [26]. Rahman et al. predict code reviewers by considering cross-project technology experience of developers [18]. Zanjani et al. presented an approach called cHRev to recommend reviewers by using previous review success of possible reviewers [29]. Xia et al. combine neighborhood methods and latent factor models to find the most suitable reviewer [25]. Xia et al. extend file path similarity method with text mining and developed an approach called TIE [24]. Kagdi et al. recommend developers for change requests with the help of feature location techniques [8]. Finally, Ouni et al. present a search-based approach using genetic algorithms [16].

Although many different types of algorithms exist to recommend appropriate reviewer for a given changeset, there is a limited set of practical tool-support to apply these algorithms in practice. Most successful example of tool support is the *mention-bot*, developed by Facebook as a GitHub bot, to identify potential reviewers of a pull request[1]. The main idea behind mention-bot is analyzing modified lines and finding a person that last touched those lines with the help of blame command of Git. Peng et al. investigate the developer experience of mention-bot and find that the bot is useful however the tool is archieved and not maintained anymore [17].

Many approaches consider change set and/or review history to recommend an appropriate reviewer. Different from the others, in this study, we create a model which also considers other software artifacts such as issues, design documents and test cases. This model requires the entire traceability data from the target project.

3 RSTRACE

A typical software development scenario is shown in Figure 1. Software artifacts, team members and relations between them are represented in this figure. In this scenario, software artifacts may include requirements, design diagrams, use case diagrams, changesets, source code files, code reviews, test cases, bugs, builds or releases. The relations could be tests, includes, reviews, commits, etc. Team members, on the other hand, may include requirements engineers, designers, developers, reviewers or testers.

In the example scenario that is described in Figure 1, requirements engineer talks to the customer first and composes Requirement #1. Design engineer draws Class diagram #2 according to the requirements. Developer implements that diagram by manipulating

[1]https://github.com/facebookarchive/mention-bot

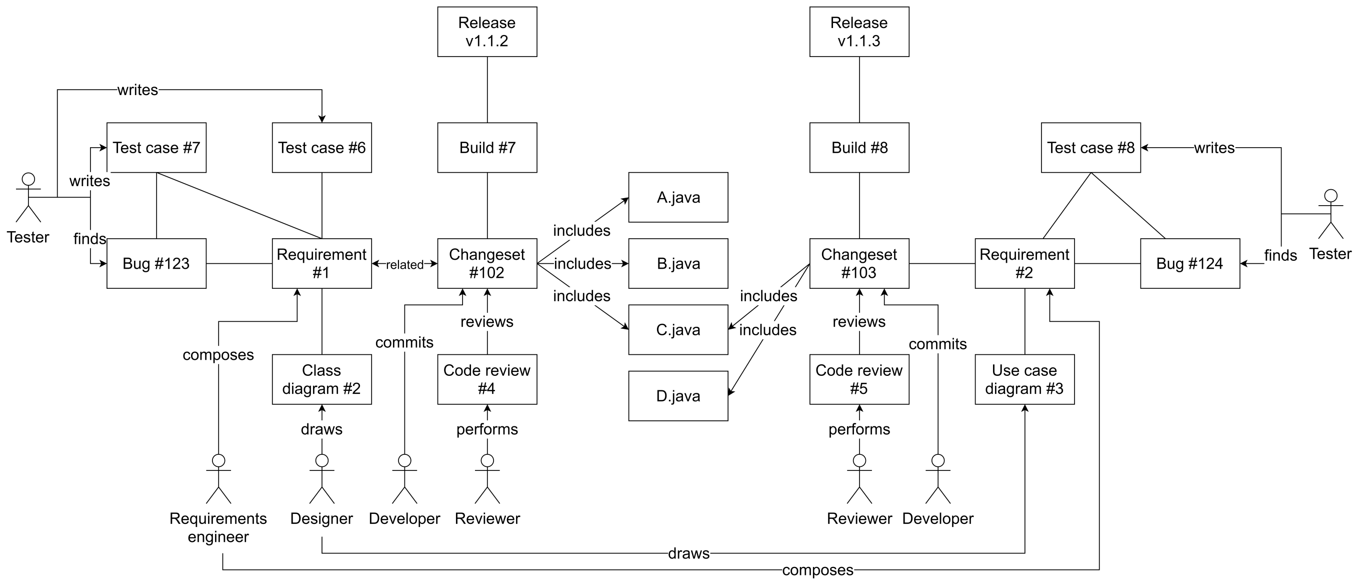


Figure 1: A sample software artifacts network

source code files A.java, B.java and C.java. Then, the developer commits those files as Changeset #102. A review is done on the changeset named Code review #4. To check if Requirement #1 is satisfied, Test case #6 is prepared. When the changeset passes Test case #6, automated build system is run and build results are ready as Build #7 and the software is released as version 1.1.2

However, a customer finds a bug related to Requirement #1 and it is recorded to bug tracking system as Bug #123 by test engineer, who also composes a new test case, Test case #7, to check if the bug is really fixed. To illustrate the complexity of a software artifact traceability graph, we provided a mirror of the described scenario on the right side of Figure 1.

With the help of a such a software artifact trabeability graph, we can visually inspect a reviewed artifact and come up with a metric that we proposed to recommend appropriate reviewers.

3.1 Visual Representation of the Reviewed Artifact

Given the example scenario described in the previous subsection, one can visualize the reviewed artifact and its neighbors. Software artifacts (e.g. requirement and changeset) and team members are the nodes; edges represent the traceability relations such as *commits*, *includes* and *reviews*. For example, in Figure 2, a developer commits a changeset, a changeset includes source code files and it is reviewed by another developer. Software artifacts graph allows developers to view how well an artifact is connected. Developers can select an artifact and analyze its impact by viewing the neighbors, while it is also possible to analyze the whole graph and infer some non-trivial information such as clusters.

Figure 2 represents the first-degree and second-degree neighbors of an example changeset. Changeset #100 has two incoming and three outgoing edges. A developer can further see the second-degree neighbors of an artifact by navigating along the graph. In

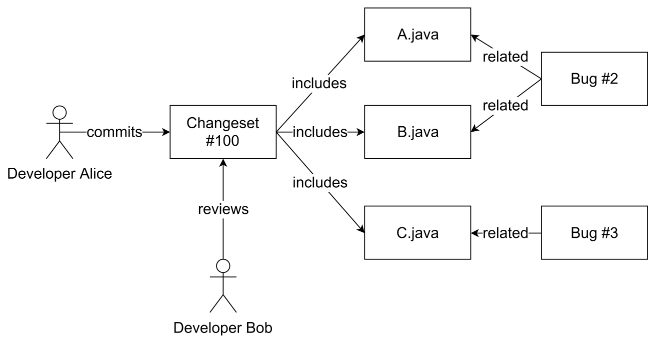


Figure 2: Visualizing the connections of an example changeset

the scenario described in Figure 2, both developer and reviewer can examine the potential impacts of a code change. This kind of a visualization augments the change in such a way that the developer can visually analyze the impact.

3.2 Finding the Appropriate Reviewer

In this study, we utilize the software artifacts graph to recommend a reviewer for a changeset. For that purpose, we define a new metric called *know-about* which intends to measure the knowledge of a developer about a source code file. It is calculated by counting all distinct paths between the developer and the source code file in the associated software artifacts graph, then summing inverse of path lengths. Thus, each developer has a know-about score for each file in the project. The formula for calculating the know-about score for a given artifact is as follows:

$$KA_{Developer-Artifact} = \sum_i \frac{1}{Length(Path_i(Developer - Artifact))}$$

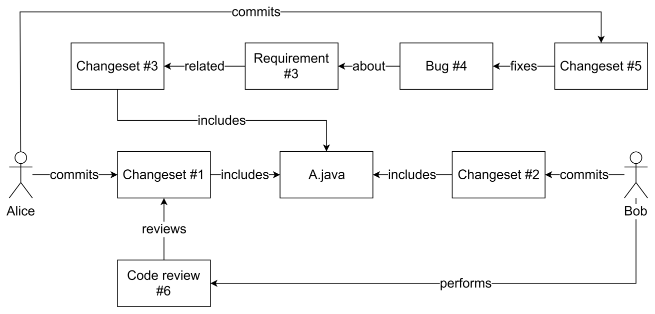


Figure 3: Relations between Alice, Bob and A.java as part of a software artifacts graph

For example, relations between two developers (Alice, Bob) and one source code file (A.java) are given in Figure 3.

According to this scenario, Alice committed a changeset that includes A.java and committed another changeset that does not directly change A.java but fixes a bug related to A.java. Similarly, Bob committed a changeset that includes A.java and reviewed another changeset that includes A.java. The *know-about* scores of Alice and Bob are calculated as follows:

Alice: There are two paths from Alice to A.java
$Length(Path_1(Alice - A.java)) = 2\ (Through\ Changeset\ \#1)$
$Length(Path_2(Alice - A.java)) = 5\ (Through\ Changeset\ \#5)$
$KA_{Alice-A.java} = 1/2 + 1/5 = 0.70$

Bob: There are two paths from Bob to A.java
$Length(Path_1(Bob - A.java)) = 2\ (Through\ Changeset\ \#2)$
$Length(Path_2(Bob - A.java)) = 3\ (Through\ Codereview\ \#6)$
$KA_{Bob-A.java} = 1/2 + 1/3 = 0.83$

When another developer commits a new changeset that would include A.java, to select the most appropriate reviewer for that change set, we need to compare the *know-about* scores for Alice and Bob. Since Bob has a higher *know-about* score, we conclude that, Bob is more suitable for reviewing a changeset that includes A.java.

This method recommends a developer only if there is a path between the developer and any file in the changeset. Therefore, if a changeset consists of files that are created for the first time, no reviewer can be recommended. The *know-about* score is proportional to the number of distinct paths available from the potential reviewer to a given artifact and inversely proportional to the path lengths.

If a changeset includes multiple files, then we calculate the know-about score for the entire changeset with the following equation:

$$KA_{Developer-Changeset} = \sum_i KA_{Developer-Artifact(i)}$$

$$where\ Artifact(i) \in Changeset$$

Where i is for each of the artifacts that a particular changeset includes. *Know-about* score of a changeset is the summation of the *know-about* scores of individual artifacts.

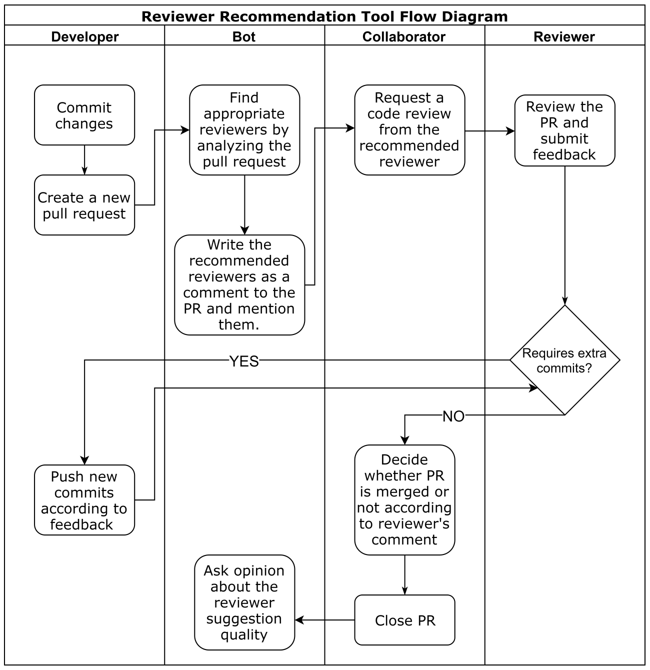

Reviewer Recommendation Tool Flow Diagram
Developer
Bot
Collaborator
Reviewer
Commit changes
Create a new pull request
Find appropriate reviewers by analyzing the pull request
Write the recommended reviewers as a comment to the PR and mention them.
Request a code review from the recommended reviewer
Review the PR and submit feedback
Requires extra commits?
YES
NO
Push new commits according to feedback
Decide whether PR is merged or not according to reviewer's comment
Close PR
Ask opinion about the reviewer suggestion quality

Figure 4: GitHub bot flow diagram

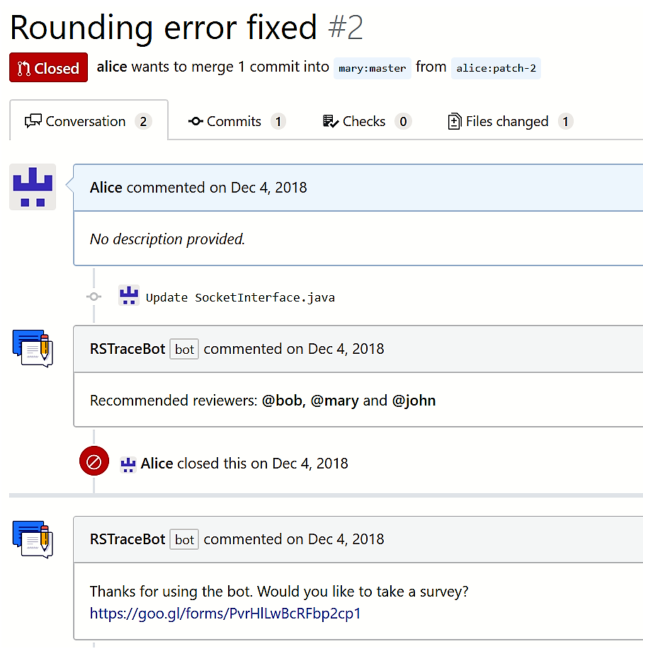
To find an appropriate reviewer for a changeset, know-about scores for each file in the changeset is summed for each team member. The developer who has the highest score excluding the developer herself is identified as the most appropriate reviewer for that changeset.

4 GITHUB BOT FOR RSTRACE

To make our algorithm usable by developers in practice for the open-source community and industry, we also implemented a GitHub bot for RSTrace, similar to mention-bot developed by Facebook. In the following sub-sections, we provide a typical workflow of a code review in GitHub, how RSTraceBot is integrated in this workflow, and the architecture of RSTraceBot.

4.1 Workflow of Code Review in GitHub

We describe a workflow of the approach of our bot integrated in a typical pull-request (PR) workflow of GitHub in Figure 4. Firstly, a developer commits some code and creates a new pull request. Then, RSTraceBot analyzes the pull request and recommends reviewers by mentioning suggested reviewers as a comment below pull request. A collaborator assigns reviewers based on the suggestion of RSTraceBot. Here our approach is non-invasive in a sense that we are only suggesting the collaborator the most appropriate reviewer names based on the RSTrace algorithm, it is the decision of collaborator to use this result or continue with the manual selection. Reviewer may approve the change or may require additional commits. When the review process is done, the pull request is either merged or not. Finally, the pull request is closed and RSTraceBot sends a survey link (to get the opinions of developers about the quality of the reviewer recommendation) below pull request. We


Rounding error fixed #2

Closed alice wants to merge 1 commit into mary:master from alice:patch-2

Conversation 2 Commits 1 Checks 0 Files changed 1

Alice commented on Dec 4, 2018

No description provided.

Update SocketInterface.java

RSTraceBot bot commented on Dec 4, 2018

Recommended reviewers: @bob, @mary and @john

Alice closed this on Dec 4, 2018

RSTraceBot bot commented on Dec 4, 2018

Thanks for using the bot. Would you like to take a survey?
https://goo.gl/forms/PvrHlLwBcRFbp2cp1

Figure 5: An example screenshot of the bot

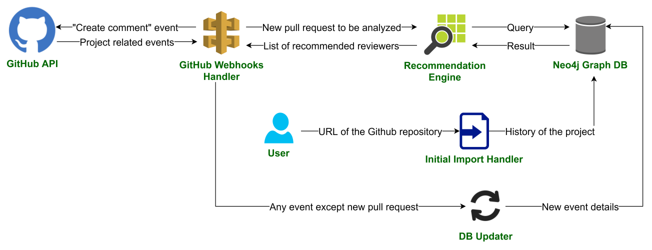


Figure 6: Modules of RSTraceBot

provide a sample screenshot from GitHub in Figure 5. The suggested number of recommended reviewers is configurable. In this example, after a pull-request is initiated, RSTraceBot recommends three reviewers.

4.2 Bot Architecture

In Figure 6, we illustrate the RSTraceBot architecture. RSTraceBot consists of five modules.

GitHub webhooks handler: Controls the communication between GitHub and the rest of RSTrace tool. When a new event happens at GitHub, it redirects the event details to database updater module. If the event is pull_request.opened, it also redirects the details to recommendation engine. Then, it writes a comment below pull request based on the response of recommendation engine.

Recommendation engine: Recommendation algorithm (RSTrace) is in this module. It gets the details of a pull request from webhooks handler, generates a recommended reviewers list and then sends the list back to webhooks handler to be written as a comment.

Table 1: Number of Artifacts according to Artifact Types

Artifact type	# of artifacts
Developer	36
Commit	942
Issue	628
Files	4151
Total	5757

Database: Stores the history of a repository in a Neo4j graph database. Recommendation engine computes the suggested reviewers by sending queries here.

Initial import handler: When a new repository starts using RSTrace, this module fetches the history of the project and imports it to the database.

Database updater: This module is responsible for updating the database after new events. For example, a new commit, issue or pull request is added to database by database updater module.

5 EXPERIMENT - QT 3D STUDIO

5.1 About the Dataset

We performed an initial experiment on Qt 3D Studio, an open source software project developed by The Qt Company. Commit history[2], issue tracking system[3] and code review system[4] are open to the public. We picked Qt 3D Studio, because it is open-source and has multiple different datasets (commit history, review history and issue history) which is inline with our methodology. In this project, Git, Jira, and Gerrit are respectively used for commit history, issue tracking and code review. We fetched the data from the servers by GrimorieLab [4] and cleaned up the data. The number of different types of artifacts are shown in Table 1. This data set was made available for other researchers at Figshare[5].

Git history contains 942 commits of 4151 files between 06/10/2017 and 06/07/2018 from 36 different developers. Each of the commits were associated with a code review. After getting the data, we parsed the commit id, date, author, modified files, reviewers and related issues. Git history also includes traceability data between commit-issue and commit-reviewer in commit message section. An example commit message is given below:

```
Fix shortcut keys in Viewer Ctrl+Shift+S for Toggle
Scale Modes Ctrl+Shift+W for Shade Mode Change-Id:
If9733d7789a85647875014c7d544b3c8ca4c6510 Task-Id:
BOUL-843 Reviewed-by: A..... M..... <a.....@kdab.com>
Reviewed-by: M..... H........ <m....@qt.io>
```

Commit messages were parsed to create traceability links between commits, issues and reviewers. The data included a total number of 628 issues. Since the data had some inconsistency for non-Latin characters in developer names, we cleaned it up by mapping to Latin ones (e.g. ä to a). Also, in some commits, developer

[2]http://code.qt.io/cgit/qt3dstudio/qt3dstudio.git/
[3]https://bugreports.qt.io/projects/QT3DS/issues
[4]https://codereview.qt-project.org/#/admin/projects/qt3dstudio/qt3dstudio
[5]https://figshare.com/s/27a35b4ae70269481a2c

Table 2: Number of Links according to Link Types

Link type	# of links
commits	942
implements	686
includes	14377
reviews	1653
Total	17658

name and surname were mixed up. These types of errors were also corrected.

5.2 Implementation

We used Neo4j graph database to store software artifacts and their relations as it allows us to store software history data as a graph easily. The following Cypher[6] query was used to calculate know-about score.

```
MATCH (a:Developer)-[r*]->(b:File)
WHERE b.id IN {fileNames}
RETURN a.id, LENGTH(r), b.id, SUM(1.0/LENGTH(r))
AS KnowAboutScore
```

fileNames indicate the modified files in a given changeset and this query returns a table that shows the know-about score of each developer for each file if it is not zero (i.e. there is at least one path between the developer and the file). Then, we sort the scores in a descending order to create an ordered recommended reviewers list.

While parsing the change history one by one, we first calculate *know-about* scores at that moment in the graph, recommend reviewers and then add nodes and relations to the graph. The final graph contains 5757 nodes and 17658 edges. Detailed breakdown of link types is given in Table 2. In this table, we have 686 instances of implements relation, 14377 instances of includes relation, and 1653 instances of reviews relation. The fact that the number of reviews relations is higher than the number of commits relations indicates that for some of the commits, there are multiple reviewers.

5.3 Visual Representation of the Impact

For a given reviewed artifact, it is important to manually inspect the possible impacted artifacts via visualization to observe its possible effects. With our approach, given an artifact it is possible to visualize the neighbors and possibly neighbors of neighbors in an iterative manner.

Figure 7 shows an example screenshot from an artifact graph database. Artifacts are colored by their category. Purple means developer, green means commit, red means file and pink means issue. The relations are marked with labels on edges. For each artifact, corresponding artifact id is written in the node, but it is abbreviated if it is too long. For example, *src/Authoring/Studio/UI/StudioAppPrefsPage.cpp* is a source code file that is at the center of Figure 7. It was modified by two commits before (starting with *21afa* and *49e31*). These two commits also have other neighbors such as other files modified by the commit. The author and the reviewers are also neighbors of the commit. Additionally, the issues that are implemented by the commits are shown in Figure 7. Such a visual representation allows a developer to see how the artifacts are directly and indirectly connected together and gives the ability to interactively assess the potential impacts to other artifacts.

[6]https://neo4j.com/cypher-graph-query-language/

5.4 Validation

To validate the accuracy of RSTrace recommendations, we use popular metrics in the literature [29] [22] [10]: Top-k Accuracy and Mean Reciprocal Rank (MRR).

The result of top-k accuracy is a percentage describing how many code reviewers were correctly recommended within top-k reviewers of the result list.

$$Top - k\ accuracy = \frac{1}{|R|} \sum_{r} isCorrect(r, k)$$

where: R is a set of reviews, isCorrect(r, k) is a function returning 1 if at least one code reviewer recommended within top k reviewers approved the review r, and 0 otherwise.

Mean Reciprocal Rank (MRR) is a value describing an average position of actual code reviewer in the list returned by recommendation algorithm.

$$MRR = \frac{1}{|R|} \sum_{r} \frac{1}{position(recommendedList(r),\ actualReviewer(r))}$$

where: R is a set of reviews, *recommendList(r)* is a function returning a sorted list of code reviewers recommended for review r, and *position(r, l)* is a function returning the rank of the code reviewer who approved review r in a sorted list l.

The maximum score of MRR is 1 if the actual reviewer is in the first position of recommended reviewers list. If the actual reviewer is not in the list, the score is 0. If more than one actual reviewer is in the list, we choose the highest MRR score. Then, overall MRR score for a project is the average of MRR scores for every review recommendation.

For the Qt 3D Studio scenario, we run RSTrace for the commits and calculate top-1, top-3, top-5 and MRR metrics. For each commit, we create a list of recommend reviewers according to historical data. If the first candidate in the recommended reviewers list is one of the actual reviewers, it is marked as true in top-1 metric. If one of the first three candidate reviewers is in the actual reviewers list, we mark it as true in top-3 metric. In some cases, a reviewer cannot be recommended due to lack of historical data for a given changeset. Therefore, they are marked as not applicable (NA) in top-k metrics. The resulting top-1, top-3, top-5 scores are shown in Table 3. The numbers represent the correctly recommended reviewers within top-k reviewers of the result list. The MRR score was calculated as 0.73 for this dataset.

6 DISCUSSION

As we have discussed in the related work section, many reviewer recommendation approaches exist in the literature. To get a more refined comparison, we provide a detailed comparison of these approaches in Table 4. The criteria used in this comparison are explained below.

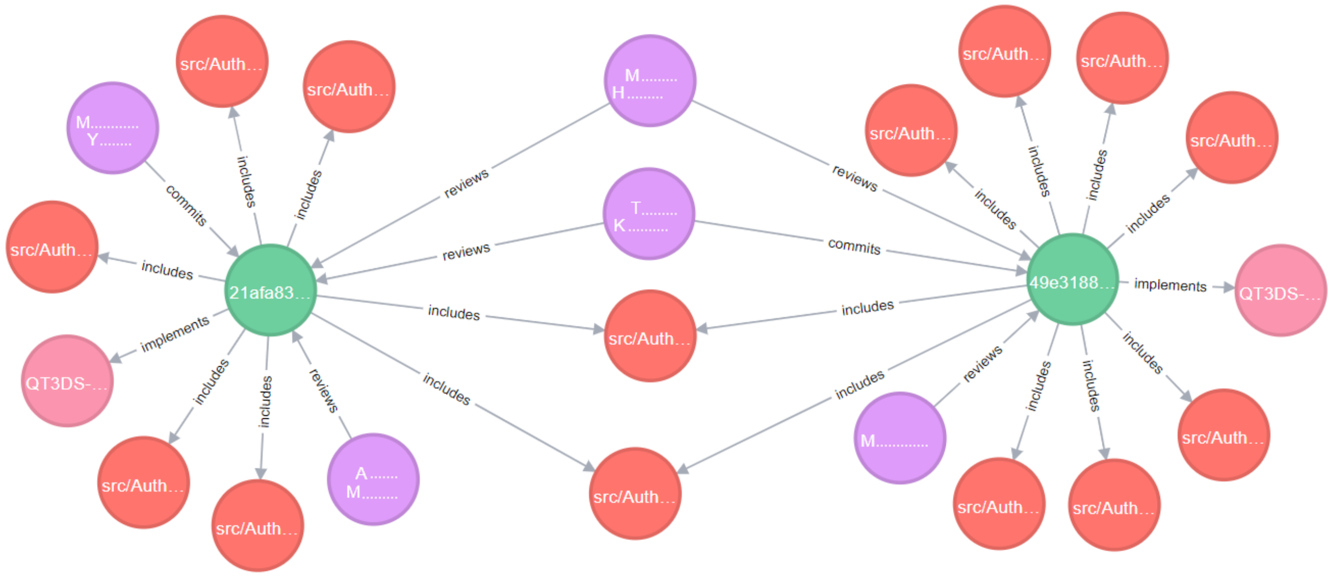


Figure 7: Second degree neighbors of a source code file. Purple means developer (names are anonymized), green means commit, red means file and pink means issue

Table 3: Top-k Metrics for Qt 3D Studio Project

	Top-1	Top-3	Top-5
True	593	800	811
False	337	130	119
NA	12	12	12
Accuracy	0.63	0.85	0.86

Algorithm/Metric Used: The different approaches proposed so far use many different types of algorithms and metrics used such as Bayesian networks [6], genetic algorithms [16], text mining [24] and support vector machines [7]. In our study, we use a metric that we have introduced called know-about to come up with best possible reviewer for a given artifact.

Open source: Indicates if the algorithm is provided as open-source or not. This attribute is essential for replicability of the algorithm.

Tool Support: Indicates if there is any tool support for practical use in practice. This attribute is important for practical usage by practitioners.

Artifact Type: Indicates the support for different types of artifacts for reviewer suggestion.

In addition to this comparison, different than the other studies, RSTrace also supports visualization of the reviewed artifact as described in Section 3.1. This would augment the developer and reviewer when analyzing the impacts of a code change.

6.1 Results

As detailed in the previous section, many reviewer suggestion approaches were proposed in the past. In this subsection, results from these approaches will be discussed and compared with RSTrace.

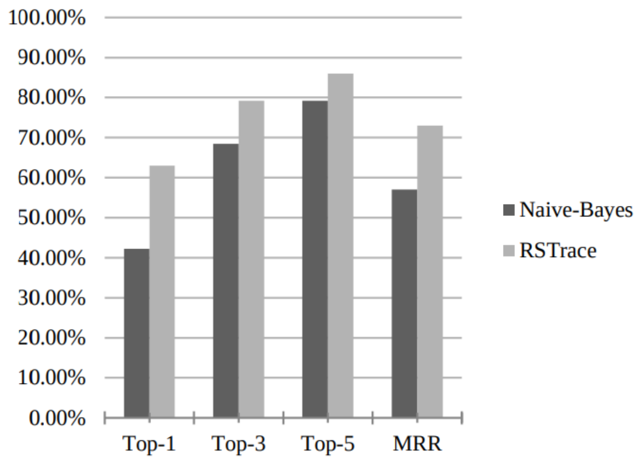


Figure 8: Comparison between RSTrace and Naive-Bayes on Qt 3D Studio Project

We implemented the Naive-Bayes algorithm that is described by Lipcak and Rossi [12]. The results of the comparison between RSTrace and Naive-Bayes is shown in Figure 8. According to the results, Naive-Bayes algorithm achieves a 42,26% top-1 accuracy where as our approach was able to achieve 63%, with a 49% improvement compare to the baseline. Similarly, RSTrace has an 15,67% improvement for top-2, and 8,59% improvement for top-3 respectively. RSTrace also has a 28% improvement in MRR scores compared to Naive-Bayes algorithm.

We also searched for publicly available datasets for further comparison. Lipcak and Rossi [12] review different approaches on code reviewer recommendation and create a dataset publicly available

Table 4: Comparison with Other Code Reviewer Suggestion Approaches

Approach	Algorithm / Metric Used	Open Source	Tool Support	Artifact Type
ReviewBot [3]	Line change history	-	+	Code
Naive-Bayes [12]	Naive-Bayes	-	-	Code
RevFinder [22]	File path similarity	+	-	Code
Correct [18]	Developer experience	-	+	Code
RevRec [26]	Hybrid of information retrieval and file location	-	-	Code
Jeong et al. [6]	Bayesian network	-	+	Code
cHRev [29]	Expertise model	-	-	Code
Developer-Source code graph [11]	Random walk algorithm	-	-	Code
TIE [24]	Text and File Location Analyses	-	-	Code
CoreDevRec [7]	Support vector machine	-	-	Code
RSTrace	Know-about metric	+	+	Any

for comparison with other approaches. The dataset includes code review information about 51 projects (37 from GitHub and 14 from Gerrit) and it is available on GitHub. We decided to ignore projects in Gerrit since as Lipcak and Rossi indicated, removal of bot reviewers from the data set seems non-trivial, and there are many pull requests with an empty list of reviewers. From this dataset set, we decided to use 37 projects that were stored in GitHub [13]. The dataset contains the commit history and reviewer history information.

Different than the Qt 3D Studio case, this dataset did not have the issue list data, which would potentially reduce the accuracy of RSTrace as it relies on different dataset types to enrich the traceability graph. We run RSTrace on 37 projects, and the results of top-5 accuracy and MRR scores per each project is shown in Figure 9. According to [12], RevFinder reaches MRR mean of 55 and Naive-Bayes with MRR mean of 61.62. According to the results, RSTrace reaches an MRR mean of 61, which is compatible with RevFinder and Naive-Bayes. The MRR results of RSTrace are much higher in Qt 3D Studio. We think this is due to the fact that in Qt 3D Studio, we also have the issue list, which enriches the traceability graph. Since we only have the MRR mean but not the MRR of individual projects, further analysis might be needed for a more accurate conclusion. Our mean top-5 for the 37 projects are 0.77, however, since mean top-k scores are not reported, we are not able to compare our scores.

6.2 Research Questions

RQ1: How does the accuracy of RSTrace compare to state-of-the-art reviewer suggestion approaches?

Approach: To address RQ1, we evaluated our results in two different data sets:

- Running RSTrace vs. Naive-Bayes algorithm in Qt 3D Studio dataset.
- Running RSTrace vs Naive-Bayes and RevFinder in 37 open-source projects described in [12]

Results: In Qt 3D Studio data, our approach has an improvement of 49% for top-1, 15,67% for top-2, and 8,59% for top-3 respectively according to the baseline approach. RSTrace also has a 28% improvement in MRR scores compare to Naive-Bayes algorithm.

We further evaluated RSTrace in 37 different open-source projects hosted on GitHub. Here we get comparable results with Naive-Bayes algorithm and RevFinder even when only considering commit history and reviewer history data.

RQ2: How does the performance of RSTrace vary according to data availability?

Approach: Since our approach works by analyzing historical data, it is possible to get less accurate results for earlier predictions. To address RQ2, we performed an experiment with Qt 3D Studio data. In this study, we execute RSTrace for every review in chronological order to obtain the lists for recommended code-reviewers. We calculate the cumulative top-k (k=1,3,5) accuracy and MRR scores per commit.

Results: We plot Figure 10 that illustrates how average top-5 score changes by time. At the end of 942 commits, it reaches to 0.86. After around the first 100 commits, top-5 score achieves 0.70, and by commit 466 top-5 score reaches 0.8.

The results show that accuracy of our approach stabilizes after a certain number of initial commits. So, in earlier stages of a project, RSTrace will not work as desired. In regard to data availability, different than the majority of the reviewer recommendation approaches, where the dataset requirement is limited to either commit history or review history, RSTrace accuracy relies on the existence of multiple different types of datasets (issue history, requirement history, build history etc.) This could be considered as a disadvantage if there are not enough datasets available.

6.3 Limitations of the Approach

Below, we discuss the limitations of our approach according to the guidelines provided in Runeson et al. [19].

Construct validity: Threats to construct validity relate to the extent to which operationalizations of a construct measure a construct as defined by a theory [19]. In our study, the main threats to construct validity are related to the suitability of our evaluation metrics. We used top-k accuracy and MRR for empirical evaluation, which are widely used metrics in the relevant literature [22] [18] [12]. Top-k accuracy and MRR metrics measure how close the set of recommended reviewers are to the actual set reviewers. Ground truth for these metrics is the set of actual reviewers. An important caveat with this assumption is the fact that the actual

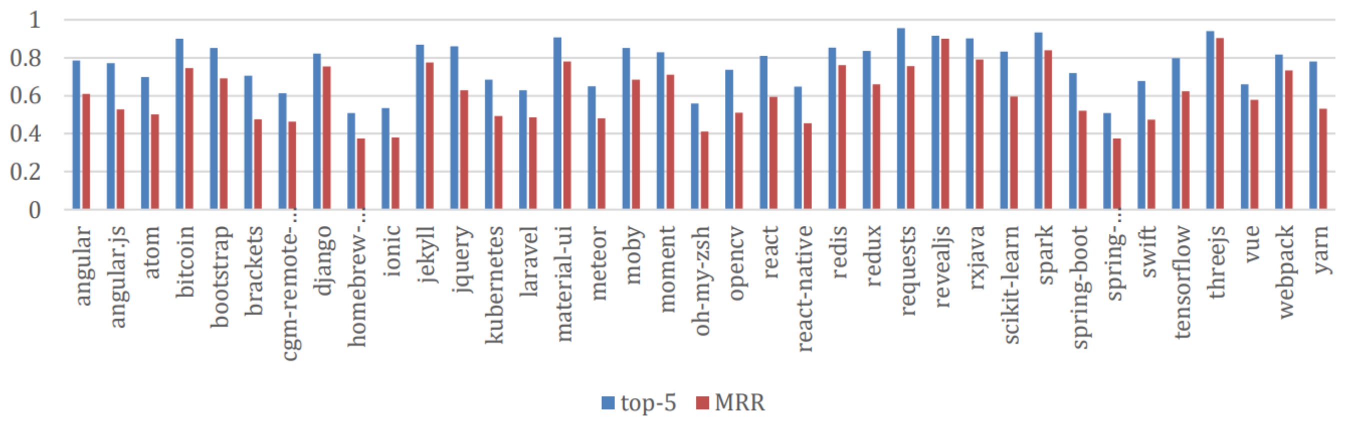


Figure 9: Top-5 and MRR scores of 37 projects hosted on GitHub when RSTrace is applied

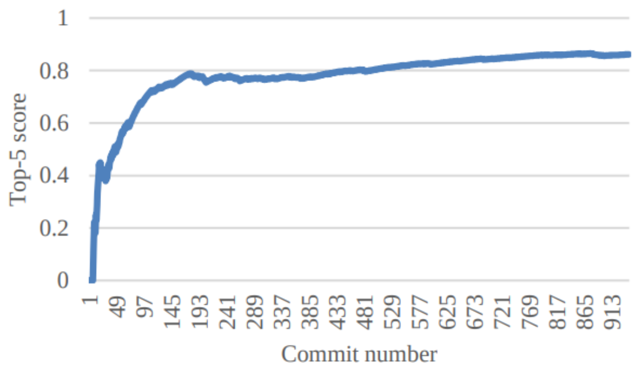


Figure 10: Commit number vs average top-5 score

set of reviewer(s) for a given commit are not necessarily the best possible reviewer(s) for the given commit. Note, however, that this applies to all other approaches in the literature.

Internal validity: Threats to internal validity are related to experimental errors and biases [19]. In this study, the main internal validity threats are related with the data collection process and implementation. To mitigate this threat, we provide source code of our implementation of both RSTrace and RSTraceBot at Figshare[7]. With regards to data collection, we used three sets of data. We initially used Qt 3D Studio data to calibrate RSTrace. We made this dataset available for other researchers as well.

In the earlier phases of the project, where a relatively few numbers of artifacts exists, the quality of the reviewer recommendation accuracy will be lower. The existing reviewer recommendation approaches also have similar limitations.

Another source of limitation is related to the number of repositories used to populate the artifacts traceability graph. Our approach is even more reliable when there are multiple types of sources of data (such as commit history, reviewer recommendation history, issue database and requirements database). The higher the number of repositories, the more complete the artifacts traceability graph will become. In Qt 3D Studio, we used three repositories (commit history, issue database and reviewer history). Our approach will potentially have higher accuracy when additional repositories exist. With the recent increased usage of DevOps tools [9], finding the variety of different datasets required by RSTrace should not be a major problem. Although the data was carefully cleaned in the repositories, there might still be a number of errors. One example of this is, in some cases the name of the authors and reviews can be represented in different formats. We have found and corrected these types of issues. However, there might be other types of similar issues.

External validity: Threats to external validity are concerned with the generalizability of our study [19]. We initially calibrated our approach using Qt 3D Studio data. We further evaluated our approach on 37 open source projects. The result of the study may not be generalized to other open source projects or commercial projects. To mitigate this issue, we prioritized evaluating our approach on a large number of projects. Other than the threats to validity, there are also few known limitations of RSTrace. For instance, in our approach, we do not consider the details of an artifact. For example, for a given commit, we do not use the number or location of lines added/removed/modified. Similarly, issues have priority information which were not utilized. We plan to address this as future work by adding weights to edges in the associated graph.

7 CONCLUSION AND FUTURE WORK

In this study, we propose a novel approach, RSTrace, that utilizes the traceability graphs of software artifacts to suggest a reviewer for a given change in an artifact. We introduce a new metric called *know-about* which measures the knowledge of a developer about an artifact by analyzing the paths from an artifact to a person in a traceability graph of artifacts. In order to evaluate RSTrace, we perform an initial experiment on Qt 3D Studio that includes 943 commits, 5757 nodes and 17658 relations. The results show that RSTrace correctly recommended 85% of reviews for top-3 recommendation. RSTrace recommends the correct code-reviewers with an MRR score of 0.73. To compare our results with other recommendation approaches, we implemented the Naive-Bayes algorithm that is described at [12] in Qt 3D Studio dataset. Our approach has an

[7]https://figshare.com/s/27a35b4ae70269481a2c

improvement of 49% for top-1, 15,67% for top-2, and 8,59% for top-3 respectively. RSTrace also has a 28% improvement in MRR scores compared to the baseline. We further ran RSTrace on 37 different open-source projects hosted on GitHub. Since these datasets only have their change history and reviewer history data available, the MRR scores were much lower compared to Qt 3D Studio, where we were able to reach an MRR score of 0.73. Still yet, even with limited datasets, we have compatible MRR scores (0.61) with the previous approaches. We summarize the main contributions of our study as follows:

- We can visualize the affected artifacts of an artifact in question and assess the potential impacts of changing the reviewed artifact.
- We provide tool support for RSTrace (RSTraceBot) which integrates our algorithm into GitHub ecosystem and make the toolset publicly available for practitioners.
- Different from other code reviewer recommendation approaches, RSTrace is not limited to recommending reviewers for only source code artifacts and can potentially be used for other types of artifacts.

We publicly share our datasets that we used to validate our study to make our study replicable. Although the initial results look promising, we still need to test RSTrace in larger data sets and industrial systems to analyze how effectively and practically RSTrace can help developers in suggesting code-reviewers. In the future, we also plan to use RSTraceBot for industrial case studies.

REFERENCES

[1] John Anvik. 2006. Automating bug report assignment. In *Proceeding of the 28th international conference on Software engineering - ICSE '06*. https://doi.org/10.1145/1134285.1134457

[2] John Anvik, Lyndon Hiew, and Gail C. Murphy. 2006. Who should fix this bug?. In *Proceeding of the 28th international conference on Software engineering - ICSE '06*. https://doi.org/10.1145/1134285.1134336

[3] Vipin Balachandran. 2013. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. *Proceedings - International Conference on Software Engineering* (2013), 931–940. https://doi.org/10.1109/ICSE.2013.6606642

[4] Santiago Dueñas, Valerio Cosentino, Gregorio Robles, and Jesus M. Gonzalez-Barahona. 2018. Perceval: Software project data at your will. In *Proceedings of the 40th International Conference on Software Engineering Companion Proceeedings - ICSE '18*, Vol. s8-I. ACM Press, New York, New York, USA, 1–4. https://doi.org/10.1145/3183440.3183475

[5] Tim Hegeman and Alexandru Iosup. 2018. Survey of Graph Analysis Applications. (2018). http://arxiv.org/abs/1807.00382

[6] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. 2009. Improving code review by predicting reviewers and acceptance of patches. *Research on Software* (2009). http://rosaec.snu.ac.kr/publish/2009/techmemo/ROSAEC-2009-006.pdf

[7] Jing Jiang, Jia Huan He, and Xue Yuan Chen. 2015. CoreDevRec: Automatic Core Member Recommendation for Contribution Evaluation. *Journal of Computer Science and Technology* 30, 5 (2015), 998–1016. https://doi.org/10.1007/s11390-015-1577-3

[8] Huzefa Kagdi, Malcom Gethers, Denys Poshyvanyk, and Maen Hammad. 2012. Assigning change requests to software developers. *Journal of software: Evolution and Process* (2012). https://doi.org/10.1002/smr.530

[9] Mik Kersten. 2018. A cambrian explosion of DevOps tools. *IEEE Software* 35, 2 (2018), 14–17. https://doi.org/10.1109/MS.2018.1661330

[10] Jungil Kim and Eunjoo Lee. 2018. Understanding review expertise of developers: A reviewer recommendation approach based on latent Dirichlet allocation. *Symmetry* 10, 4 (2018), 5–7. https://doi.org/10.3390/sym10040114

[11] John Boaz Lee, Akinori Ihara, Akito Monden, and Ken Ichi Matsumoto. 2013. Patch reviewer recommendation in OSS projects. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC* 2 (2013), 1–6. https://doi.org/10.1109/APSEC.2013.103

[12] Jakub Lipčák and Bruno Rossi. 2018. A Large-Scale Study on Source Code Reviewer Recommendation. In *44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE*. https://doi.org/10.1109/SEAA.2018.00068

[13] Jakub Lipčák and Bruno Rossi. 2018. Rev-rec Source - Code Reviews Dataset (SEAA2018). (2018). https://doi.org/10.6084/m9.figshare.6462380.v1

[14] Laura MacLeod, Michaela Greiler, Margaret Anne Storey, Christian Bird, and Jacek Czerwonka. 2018. Code Reviewing in the Trenches: Challenges and Best Practices. *IEEE Software* (2018). https://doi.org/10.1109/MS.2017.265100500

[15] Hoda Naguib, Nitesh Narayan, Bernd Brügge, and Dina Helal. 2013. Bug report assignee recommendation using activity profiles. In *IEEE International Working Conference on Mining Software Repositories*. https://doi.org/10.1109/MSR.2013.6623999

[16] Ali Ouni, Raula Gaikovina Kula, and Katsuro Inoue. 2017. Search-based peer reviewers recommendation in modern code review. *Proceedings - 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016* (2017), 367–377. https://doi.org/10.1109/ICSME.2016.65

[17] Zhenhui Peng, Jeehoon Yoo, Meng Xia, Sunghun Kim, and Xiaojuan Ma. 2018. Exploring How Software Developers Work with Mention Bot in GitHub. *Proceedings of the Sixth International Symposium of Chinese CHI* (2018), 152–155. https://doi.org/10.1145/3202667.3202694

[18] Mohammad Masudur Rahman, Chanchal K. Roy, and Jason A. Collins. 2016. CoRReCT. *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16* (2016). https://doi.org/10.1145/2889160.2889244

[19] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* (2009). https://doi.org/10.1007/s10664-008-9102-8 arXiv:gr-qc/9809069v1

[20] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review. *Proceedings of the 40th International Conference on Software Engineering Software Engineering in Practice - ICSE-SEIP '18* (2018), 181–190. https://doi.org/10.1145/3183519.3183525

[21] Emre Sülün. 2019. Suggesting Reviewers of Software Artifacts using Traceability Graphs. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 19), August 26-30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA. https://doi.org/10.1145/3338906.3342507

[22] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken Ichi Matsumoto. 2015. Who should review my code? A file location-based code-reviewer recommendation approach for Modern Code Review. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings* (2015), 141–150. https://doi.org/10.1109/SANER.2015.7081824

[23] Eray Tüzün, Bedir Tekinerdogan, Yagup Macit, and Kursat Ince. 2019. Adopting integrated application lifecycle management within a large-scale software company: An action research approach. *Journal of Systems and Software* 149 (2019), 63–82. https://doi.org/10.1016/J.JSS.2018.11.021

[24] Xin Xia, David Lo, Xinyu Wang, and Xiaohu Yang. 2015. Who should review this change?: Putting text and file location analyses together for more accurate recommendations. *2015 IEEE 31st International Conference on Software Maintenance and Evolution, ICSME 2015 - Proceedings* (2015), 261–270. https://doi.org/10.1109/ICSM.2015.7332472

[25] Zhenglin Xia, Hailong Sun, Jing Jiang, Xu Wang, and Xudong Liu. 2017. A hybrid approach to code reviewer recommendation with collaborative filtering. *SoftwareMining 2017 - Proceedings of the 2017 6th IEEE/ACM International Workshop on Software Mining, co-located with ASE 2017* (2017), 24–31. https://doi.org/10.1109/SOFTWAREMINING.2017.8100850

[26] Cheng Yang, Xun hui Zhang, Ling bin Zeng, Qiang Fan, Tao Wang, Yue Yu, Gang Yin, and Huai min Wang. 2018. RevRec: A two-layer reviewer recommendation algorithm in pull-based development model. *Journal of Central South University* 25, 5 (2018), 1129–1143. https://doi.org/10.1007/s11771-018-3812-x

[27] Xin Yang. 2014. Social Network Analysis in Open Source Software Peer Review. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. https://doi.org/10.1145/2635868.2661682

[28] Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. 2016. Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment? *Information and Software Technology* 74 (2016), 204–218. https://doi.org/10.1016/j.infsof.2016.01.004

[29] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. 2016. Automatically Recommending Peer Reviewers in Modern Code Review. *IEEE Transactions on Software Engineering* 42, 6 (2016), 530–543. https://doi.org/10.1109/TSE.2015.2500238

[30] Wei Qiang Zhang, Li Ming Nie, He Jiang, Zhen Yu Chen, and Jia Liu. 2014. Developer social networks in software engineering: construction, analysis, and applications. *Science China Information Sciences* 57, 12 (2014), 1–23. https://doi.org/10.1007/s11432-014-5221-6

[31] Thomas Zimmermann and Nachiappan Nagappan. 2008. Predicting defects using network analysis on dependency graphs. *Proceedings of the 13th international conference on Software engineering - ICSE '08* (2008), 531. https://doi.org/10.1145/1368088.1368161