ANALYZING DEVELOPER CONTRIBUTIONS USING ARTIFACT TRACEABILITY GRAPHS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
Hamdi Alperen Çetin
December 2020

Analyzing Developer Contributions using Artifact Traceability Graphs
By Hamdi Alperen Çetin
December 2020

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Eray Tüzün(Advisor)

Uğur Doğrusöz

Mehmet Akşit

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

ANALYZING DEVELOPER CONTRIBUTIONS USING ARTIFACT TRACEABILITY GRAPHS

Hamdi Alperen Çetin
M.S. in Computer Engineering
Advisor: Eray Tüzün
December 2020

Software artifacts are the by-products of the development process. Throughout the life cycle of a project, developers produce different artifacts such as source files and bug reports. To analyze developer contributions, we construct artifact traceability graphs with these artifacts and their relations using the data from software development and collaboration tools.

Developers are the main resource to build and maintain software projects. Since they keep the knowledge of the projects, developer turnover is a critical risk for software projects. From different viewpoints, some developers can be valuable and indispensable for the project. They are the key developers of the project, and identifying them is a crucial task for managerial decisions. Regardless of whether they are key developers or not, when developers leave the project, their work should be transferred to other developers. Even though all developers continue to work on the project, the knowledge distribution can be imbalanced among developers. Evaluating knowledge distribution is important since it might be an early warning for future problems.

We employ algorithms on artifact traceability graphs to identify key developers, recommend replacements for leaving developers and evaluate knowledge distribution among developers. We conduct experiments on six open source projects: Hadoop, Hive, Pig, HBase, Derby and Zookeeper. Then, we demonstrate that the identified key developers match the top commenters up to 98%, recommended replacements are correct up to 91% and identified knowledge distribution labels are compatible with the baseline approach up to 94%.

Keywords: key developers, social networks, artifact traceability graphs, developer replacement, developer turnover, knowledge distribution.

ÖZET

YAPI İZLENEBİLİRLİK ÇİZGELERİ KULLANARAK GELİŞTİRİCİ KATKILARINI ANALİZ ETME

Hamdi Alperen Çetin
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Eray Tüzün
Aralık 2020

Yazılım yapıları geliştirme sürecinin yan ürünleridir. Geliştiriciler projenin hayat döngüsü boyunca kaynak dosyaları ve hata raporları gibi farklı yapılar üretirler. Biz yazılım geliştirme ve işbirliği araçlarındaki veriyi kullanarak, yapılar ve aralarındaki bağlantılar ile yapı izlenebilirlik çizgeleri oluşturduk.

Geliştiriciler bir yazılım projesini geliştirme ve sürdürme sürecinde kullanılan asıl kaynaktırlar. Geliştiriciler projelerin bilgisine sahip oldukları için geliştirici devri yazılım projeleri için kritik bir risktir. Bazı geliştiriciler farklı bakış açılarından proje için değerli ve vazgeçilmez olabilir. Onlar projenin anahtar geliştiricileridir ve onları belirlemek yönetimsel kararlar için çok önemlidir. Anahtar geliştirici olsun veya olmasın, geliştiriciler projeden ayrıldığında işleri başka geliştiricilere aktarılmalıdır. Bütün geliştiriciler çalışmaya devam etse bile, geliştiriciler arasındaki bilgi dağılımı dengesiz olabilir. Bilgi dağılımını değerlendirmek gelecekteki problemler için erken bir uyarı olabileceğinden önemlidir.

Biz anahtar geliştiricileri belirlemek, ayrılan geliştirici yerine geliştiriciler önermek ve takımdaki bilgi dağılımını değerlendirmek için yapı izlenebilirlik çizgeleri üzerinde algoritmalar kullandık. Hadoop, Hive, Pig, HBase, Derby ve Zookeeper isimli altı açık kaynak proje ile deneyler yaptık. Sonrasında, anahtar geliştiricileri tanımlamada %98'e varan, ayrılan geliştiriciler için geliştirici önermede %91'e varan doğrulukta sonuçlar elde ettik ve bilgi dağılımı için kullandığımız etiketler %94'e varan oranda temel yöntem ile uyumlu çıktı.

Anahtar sözcükler: Anahtar geliştiriciler, sosyal ağlar, yapı izlenebilirlik çizgeleri, geliştirici değiştirme, geliştirici devri, bilgi dağılımı.

Acknowledgement

First and foremost, I would like to express my gratitude to my advisor Asst. Prof. Eray Tüzün for his guidance throughout my master's studies and steering me in the right direction whenever I needed. The door to Prof. Tüzün was always open whenever I had a question or a problem with my research.

I also would like to thank the jury members, Prof. Uğur Doğrusöz and Prof. Mehmet Akşit, for spending time to read my thesis and accepting to be on the committee.

I would like to thank Emre Doğan, Barış Ardıç and other BILSEN *(Bilkent University Software Engineering and Data Analytics Research Group)* members for helping my studies with their valuable ideas and comments. Also, I would like to thank my office mates at EA-527. I feel very lucky to have great memories with them until the pandemic breakdown.

Last but not least, I would like to express my deepest gratitude to my family for their unconditional support with love and understanding. Also, I would like to thank everyone I am yet to mention.

In memory of my studies during the coronavirus pandemic.

December 2020, Ankara

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software artifacts are produced throughout the development process. There is a multitude of different types of software artifacts such as issues, design documents, data models and source files. Conway [2] claimed "organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations" (also known as Conway's Law). Also, Herbsleb and Grinter [3] examined multi-site development and stated that simple daily routines like meeting in hallways and finding the right person to ask about a problem are an indispensable part of the work. Therefore, the structure of the project reflects the connections and communication among artifacts and developers.

In a typical development environment, software development and collaboration tools are widely used. Artifacts and their relations can be extracted from these tools (e.g., version control systems and issue tracking systems). Graphs are useful basic data types to keep data points and relations among them. In order to keep the structure of the organizations working on the projects, we construct artifact traceability graphs representing artifacts as nodes and relations as edges. Since software development mainly depends on human effort, developers are the most important resource to build and maintain software projects. Therefore, we study different perspectives by analyzing developer contributions.

1.1 Research Problem

In a project, some developers take more responsibility, and the success rate of the project heavily depends on these developers. Thus, they are valuable and essential to develop and maintain the project, in other words, they are the *key developers* of the project. Developers leave and join projects due to numerous reasons such as transferring to another project in the same company or leaving a company to work in another one. When developers leave the project, they should be replaced by other developers in a short period of time. This is also known as developer turnover, which is a common phenomenon in software development. For instance, median tenure at Google and Amazon are 1.1 years and 1 year respectively according to a report published at Payscale[1]. Also, it is a critical risk for software projects [4]. It is more critical when the key developers leave the project. Therefore, identifying valuable and indispensable developers is a vital and challenging task for project management.

All developers contribute to the project in various tasks, thus developers can be valuable for the project in many different ways. For instance, a developer may know a specific module very well, while another one knows a little related to multiple modules. In our study, similar to our previous works [5, 6], we examine key developers under three categories: *jacks*, *mavens* and *connectors*.

Our motivation for this categorization comes from *The Tipping Point* by Gladwell [7]. The book discusses the reasons behind word-of-mouth epidemics. In *The Law of Few* chapter, the author justifies that three kinds of people turn ideas into epidemics and they are responsible for tipping ideas: connector, salesman and maven. Connectors have connections to different social groups, and they allow ideas to spread between these groups. Salesmen have a charisma that allows them to persuade people and change their decisions. Mavens have a great knowledge of specific topics and thus help people to make informed decisions.

Since there are traceable links among software artifacts, and they mirror the

[1]https://www.payscale.com/data-packages/employee-loyalty/least-loyal-employees (Accessed on 24 Sep 2020)

connections among people in real life, we propose to use a similar categorization, *connector* and *maven*, as described in the book [7] to find the key developers in a software project. A typical *connector* represents a developer who is involved in different (sub)projects or different groups of developers. Connecting divergent groups or (sub)projects increases this type of developers' significance because they connect the developers who are not in the same group (i.e., team) and touching different parts of a project means collective knowledge from different aspects of the project. The *maven* category represents the developers who are masters in details of specific modules or files in the project. Being the rare experts of specific parts of the source code makes these developers difficult to replace.

Jacks (of all trades) are the developers who have a broad knowledge of the project. They use or modify files from different parts of the project. Here, *jack* and *connector* definitions may interfere with each other since both define key developers who touch different parts of the project. To make it more clear, the *jack* category purely focuses on knowledge when the *connector* category focuses on connecting developers. "Jack" name comes from a figure of speech, jack of all trades, to define people "who can do passable work at various tasks"[2]. For the developers who have a broad knowledge of projects, we use *jack* to remind of this phrase.

As mentioned above, developer turnover is a critical risk for software projects [4]. Regardless of whether a key developer or not, the work should be transferred to another developer when a developer leaves the project. Stakeholders of the software project should handle such cases with a minimal negative effect on the development. Using the traceable links between software artifacts, the best replacements for leaving developers can be found. These replacements can take over all the work or can mentor newcomer developers in their learning process of the jobs of the leaving developer.

The risks mentioned above do not just occur due to developer turnover. Even though all the developers continue to work on the project, the knowledge distribution can be imbalanced among developers. In that case, the project depends on a

[2]https://www.merriam-webster.com/dictionary/jack-of-all-trades (Accessed on 24 Sep 2020)

very small group of developers (some of them may be key developers). Foreseeing such cases are possible and considerably important for managerial decisions.

To discover the points mentioned above, in this study, we address the following research questions (RQs):

RQ 1: How can we identify key developers in a software project?

RQ 1.1: How can we identify jacks in a software project?

RQ 1.2: How can we identify mavens in a software project?

RQ 1.3: How can we identify connectors in a software project?

RQ 2: How can we find replacements (successors) for leaving developers?

RQ 3: How can we evaluate knowledge distribution in a software project?

1.2 Contributions

The main contributions of this thesis are:

- By using artifact traceability graphs, we offer:
  - A novel categorization of the key developers and algorithms to identify the developers for each category.
  - An algorithm to recommend replacements (successors) for leaving developers.
  - An algorithm to evaluate knowledge distribution in development teams.
- We provide a proof of concept tool presenting tables and charts for the RQs. It visualizes the generated results of the experiments and shows how a manager dashboard tool using our algorithms would seem like.

In the following chapter, we share related work. In Chapter 3, we explain our methodology addressing the RQs. In Chapter 4, we share the details of the datasets and the important points of the preprocessing. In Chapter 5, we perform case studies in six different open source software (OSS) projects. In Chapter 6, we present a proof of concept tool for traversing our results day by day. In Chapter 7, we discuss the RQs, scalability of the algorithms and practical impacts of the study. In Chapter 8, we discuss the threats to validity of our study. In Chapter 9, we present our conclusions and possible future works.

Chapter 2

Related Work

In the literature, there are many studies on truck factor, developer recommendation, developer social networks and developer roles/types. In the following, we present them under separate sections.

2.1 Truck Factor

Truck factor (i.e., bus factor) is the answer to the following question: *What is the minimum number of developers who have to leave the project before the project becomes incapacitated and has serious problems?* To address this problem, Avelino et al. [8] associated files to authors by using the degree of authorship [9], then they found the minimum number of developers whose total file coverage is more than 50% of all files. Cosentino et al. [10] measured developers' knowledge on artifacts (e.g., files, directories and project itself) with different metrics such as "last change takes it all" and "multiple changes equally considered". They defined primary and secondary developers for the artifacts and proposed that the project will have problems with the artifact if all primary and secondary developers leave the project. Rigby et al. [11] studied a model on file abandonment. In their study, the author of a line is assigned by using *git blame*, and a file is abandoned

when the authors of 90% of its lines left the project. They proposed to remove developers randomly until a specific amount of file loss occurs, and use the number of removed developers as the truck factor at that point.

Moreover, some researchers published empirical studies on existing truck factor algorithms. Avelino et al. [12] investigated abandoned OSS projects. In their definition, a project is abandoned when all truck factor developers leave. Ferreira et al. [13] performed a comparative study on truck factor algorithms and made a comprehensive discussion on them from many different viewpoints such as the accuracy of the reported results in the studies and the reasons why the truck factor algorithms fail in some circumstances.

2.2 Developer Recommendation

Developer recommendation has been a hot software engineering research topic. Recommending proper developers for bug resolution, code reviewers for new code changes and successors to replace leaving developers are some of the application areas of developer recommendation.

Xia et al. [14] proposed a model suggesting developers for bug reports. Their method is a linear combination of scores from bug-report-based analysis and developer-based analysis. These analyses use terms and topics in the bug reports, affected products and affected components. Balachandran [15] presented a code reviewer recommendation model based on line change history. The model returns a list of developers according to their scores calculated heuristically on line change history of pull requests. Canfora et al. [16] proposed an approach, YODA, to recommend mentors to the newcomers.

Rigby et al. [11] proposed a model for suggesting successors of leaving developers. For abandoned files, their model suggests five potential successors by considering the number of files that are co-changed with abandoned files. Also,

the model suggests one new developer who has no co-change experience. To evaluate their approach, they used another model that suggests developers randomly, and they compared the results. Nassif and Robillard [17] replicated the study of Rigby et al. [11]. Other than validating the results shared by the first study, they also extended the previous study by adding weights to files and conducting experiments on different time periods.

2.3 Developer Social Networks

Many studies have been published on developer social networks [18]. Wu and Goh [19] studied the long term effects of communication patterns on success. They performed experiments on how graph centrality, graph density and leadership centrality affect the success of OSS projects. Also, Kakimoto et al. [20] worked on knowledge collaboration through communication tools. They applied social network analysis to four OSS communities, and partially verified their hypothesis, which claims "Communications are actively encouraged before/after OSS released, especially among community members with a variety of roles but not among particular members" [20]. Also, Joblin et al. [21] worked on network-based metrics (e.g., degree centrality) while investigating core and peripheral developers. Moreover, Allaho and Lee [22] conducted a social network analysis on OSS projects and found that OSS social networks follow a power-law distribution which means a small number of developers dominate the projects.

2.4 Developer Roles and Types

There have been a number of studies examining developer types from different perspectives. Kosti et al. [23] investigated archetypal personalities of software engineers. They chose extraversion and conscientiousness as their main criteria and focused on the binary combinations of them. Cheng and Guo [24] made an activity-based analysis of OSS contributors, then adopted a data-driven approach

to find out the dynamics and roles of the contributors. Milewicz, Pinto and Rodeghero [25] worked on the contributor roles in scientific OSS projects. They classified researchers as senior, junior and thirdparty.

Ortu et al. [26] inspected GitHub contributors as users (contributors without a commit) and developers (contributors who have source code commits). Then, they inspected developers as one-commit developers and multi-commit developers. By examining the received/sent comments of the contributors, they found that different groups play different roles and have different communication patterns (e.g., level of politeness).

In the literature, there are studies examining core and periphery [27, 21], core, active, occasional and rare [28], core, external and mutant [29], key [30], hero [31] and elite [32] developers in OSS projects. Likewise, Zhou and Mockus [33] claimed that Long Term Contributors (LTCs) are valuable for projects. These developer definitions (e.g., core and key) all have similar definitions and are closely related to our study.

Goeminne and Mens [34] examined the activity distribution in three OSS communities. They found that the contributions to commits, mailing list and bug reports are imbalanced and the Pareto principle (20% of the contributors make 80% of the contributions) holds for the activities of *committers*, *mailers* and *bug report changers*. Also, for each project, they reported Venn diagrams for these three contributor types and showed the overlaps between the top 20 contributors. In the reported diagrams, a small number of developers (2, 4 and 0) are at the intersection of all three contributor types. This shows that different contributors are active on different platforms.

Yamashita et al. [35] studied the Pareto principle in 2496 OSS projects. They found that 47% of the projects are Pareto compliant in commit-based evaluation. By referring to the study of Yamashita et al. [35], Agrawal et al. [31] worked on hero developers in public and enterprise GitHub software projects. According to their definition, a project has hero developers if 20% of the developers made 80% of the contributions. Then, they analyzed hero and non-hero projects and

claimed that the hero developers are very common particularly in medium to large projects and organizations should work on keeping this type of developers.

In another study, Yamashita et al. [36] classified the OSS projects in a different way. According to attracting new developers and retaining the existing ones, they classified the projects as magnet or sticky. They found that the sticky and magnet values of the projects change in time, and they shared the likelihood of those transactions. Since developers need time to become an indispensable participant (i.e., key developer) of the team and sticky projects are inclined to keep existing developers, key developers may exist more in sticky projects. However, this aspect needs further inspection.

Oliva et al. [30] worked on characterizing the key developers, that is "the set of developers who evolve the technical core". First, they detected the core commits by constructing a call-graph of the classes in the project. Then, they ranked the developers according to their core commit counts and considered the developers who made roughly 80% of the core commits as the key developers. They did not share any validation for the identified key developers (e.g., making a developer survey or using another data source showing similar results). Afterward, they analyzed the identified key developers in terms of contribution characteristics, communication and coordination within the project on a small project with 16 developers.

Bella, Sillitti and Succi [28] clustered OSS contributors by using the k-means algorithm with the features from version control system such as *number of commits* made by the developer, *number of files* edited by the developer and *days in the project* of the developer. After the clustering analysis, they classified OSS contributors as core, active, occasional and rare developers. It is an onion-like structure. Core developers imply a small group of developers who develop most of the project and make the most important contributions for a long period of time. Active developers also contribute to different specific parts and handle important jobs for a long time. Occasional developers refer to a large number of developers contributing to a limited number of specific files occasionally. Rare developers contribute to the project in limited time periods, then they stop contributing.

Crowston et al. [27] examined the core and periphery of OSS team communications. They analyzed if the following three methods produce similar results or not: the contributors named officially (e.g., support manager and developer labels extracted from SourceForge[1]), the contributors who contribute the most to the bug reports, and the contributors who are defined by a pattern of interactions in bug tracking systems.

Joblin et al.[21] studied core and peripheral developers. The core developers are the essential developers in the projects as the key developers in our study. They worked on count-based (e.g., number of commits as in [31]) and network-based (e.g., degree centrality in developer graph from version control systems and mailing lists) metrics. They established a ground truth by making a survey with 166 participants. We were not able to examine their data because the project and survey data links are not accessible through their website[2].

Padhye, Mani and Sinha [29] analyzed commits and developers (i.e., committers) in the *fork-and-pull* model of GitHub. They classified commits and developers as core, external and mutant. Core committers have access to the repository of the project, and the commits made directly to the main repository are named as core commits. External commits are made through a pull request and accepted by core committers, and the author of these kinds of commits are considered as external committers. The mutant category refers to the commits made in rejected pull requests or the commits made for personal uses. Then, they shared some statistics about these definitions for the 89 top-starred GitHub projects and their forks. For example, they shared the distributions of the community sizes for core, external and mutant categories. They defined core developers in an access-based manner (access to the repository). Similarly, Wang et al. [32] checked whether developers have write permission to the repository or not while identifying elite developers. Then, they revealed elite developers' activities. For example, elite developers manage supportive and communicative activities more when the project grows.

[1]https://sourceforge.net/ (Accessed on 24 Sep 2020)
[2]http://siemens.github.io/codeface/icse2017/ (Accessed on 24 Sep 2020)

Zhou and Mockus [33] defined an LTC as "a participant who stays with the project for at least three years and is productive". They claimed that LTCs are crucial for the success of the projects. They mainly investigated how a new joiner becomes an LTC (i.e., a valuable contributor).

Chapter 3

Methodology

To address the RQs, we first construct an artifact traceability graph of the project by taking recency into account in Section 3.1. Then, we propose separate algorithms for key developer types in Section 3.2 through Section 3.4. We propose an approach to determine threshold scores for identifying the significant set of key developers in each key developer category in Section 3.5. Afterward, we propose an algorithm to recommend replacements for leaving developers in Section 3.6 and another algorithm to evaluate the knowledge distribution in development teams in Section 3.7.

3.1 Artifact Traceability Graph

Artifact traceability graphs include software artifacts and the connections between them. We denote nodes for software artifacts, which are developers, change sets (e.g., commits in Git), source files and issues. Then, we denote undirected edges for the relations (e.g., commit, review, include and linked) between those artifacts. For example, we add an edge for a *commit* relation between the developer node and the change set node if the developer is the author of the change set. The edges are undirected because reaching from one change set to another

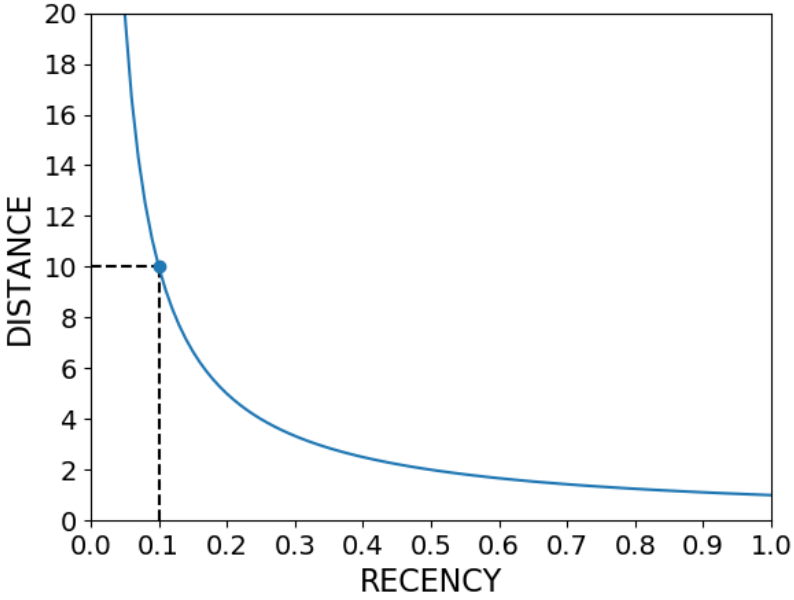


Figure 3.1: Distance against recency

should be possible over the edges if they include the same files (i.e., co-changed files). Developers *commit* or *review* change sets. Change sets *include* a set of source files. Issues can be *linked* to a set of change sets and vice versa.

In the graph, we denote distances for each edge. Distances of the edges between developers and change sets are always zero (0) because these connections are there in order to keep track of who made *commit*s and *review*s. Other than *commit* and *review* cases, edge distances are calculated by using the recency of the bound change set. Our distance metric is inversely proportional to the recency of the change set as shown in Fig. 3.1. Recency and distance metrics are calculated as follows:

$$Recency = 1 - \frac{\text{\# of days passed}}{\text{\# of days in the graph}} \tag{3.1}$$

$$Distance = \frac{1}{Recency} \tag{3.2}$$

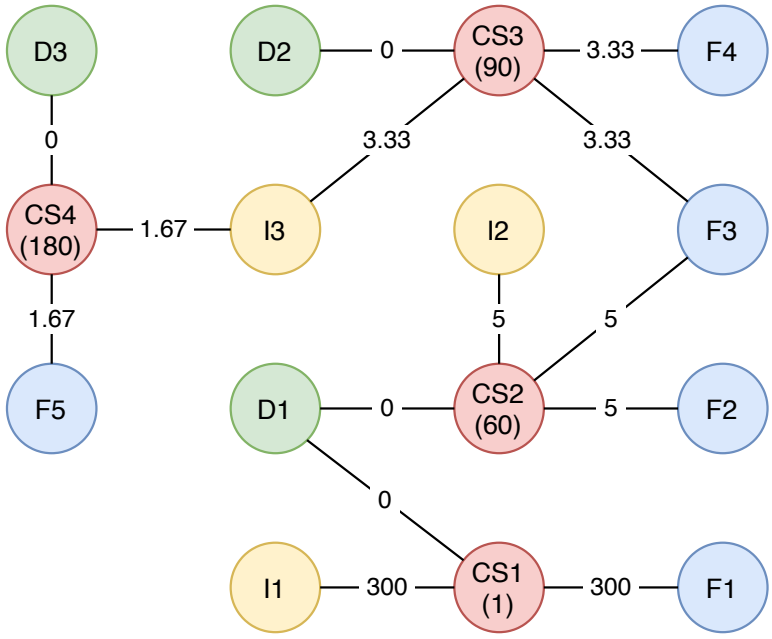


Figure 3.2: Sample artifact traceability graph. (D: Developer, F: File, CS: Change Set, I: Issue)

Fig. 3.2 shows a sample traceability graph, where the graph includes 300 days of the project history, and the numbers in the parentheses are the days that the commits are made. For example, CS3 was committed on the 90^{th} day (i.e., 210 days ago). All the edges of CS3 have the same distance, which is calculated as follows.

$$Distance = \frac{1}{Recency} = \frac{1}{1 - \frac{210}{300}} = \frac{1}{0.30} = 3.33$$

We add distance to the edges in the graph since assuming that all edges having the same importance might be problematic since some edges represent recent commits and reviews while others represent older ones. Our recency and distance definitions are utilized here to distinguish these types of situations. For example, the distance between CS3 and F4 is 3.33 while the distance between CS4 and F5 is 1.67, and there are around three months between the commit times of CS3 and CS4.

3.2 Jacks (RQ 1.1)

To find Jacks in a software project, we analyze the general knowledge of the developers on the project. By looking at the history of the project from its version control data, we can say that the source files keep the knowledge, in other words, the know-how of the project. There are studies to find the authors of source files (e.g., degree of authorship [9]). Authorship is not only about being the first author of the file but also about changing the source files in time depending on the recency of the change. In our study, we define *reachability* similar to this definition. If developers can reach a file, they know that file. Also, multiple developers can reach the same file at the same time. In the following, we explain how we find reachable files and file coverage for each developer.

3.2.1 Finding Reachable Files

We define reachable files of a developer as the files that are reached by the developer through the connections in the artifact traceability graph. For example, in Fig. 3.2, the D2 node can reach every file node in the graph through change sets, issues and other developers if there is no distance limit (i.e., a limit for the sum of distances on the edges in the graph). Actually, every developer can reach every file if the graph is connected and there is no distance limit. In that case, every developer would know every file, and we could not distinguish which developers know which source files. Therefore, to handle these situations, we define the following rules:

1. We need to set a threshold (limit) for distance while reaching from developer nodes to file nodes. For example, in Fig. 3.3, D2 cannot reach F5 if the threshold is 5 because $3.33 + 1.67 + 1.67 = 6.67$ and 6.67 is beyond the threshold 5.

2. One developer cannot reach files through other developers because it would transfer reachable files of a developer to another developer if the distance

threshold is large enough. For example, in Fig. 3.3, D2 cannot reach F1 through D1, even if the distance threshold is 308.33 or more.

Distance threshold is a parameter, and it depends on the distance formula given in Equation (3.2). Due to its nature, distance goes to infinity when recency goes to zero as shown in Fig. 3.1. In the sample graph, the oldest relations are the relations from the first day, and their edges have the highest distance. In this case, their distance is calculated as follows:

$$Distance = \frac{1}{Recency} = \frac{1}{1 - \frac{299}{300}} = \frac{1}{\frac{1}{300}} = 300$$

Therefore, we need to set our threshold to 300 if we want to use every direct relation in the graph. Since almost all recently changed files are reachable by the developers who have recent commits in that case, using 300 as the threshold would not enable us to distinguish which developers know which files.

We follow a simple way while deciding the distance threshold. In a 300-day graph, the edges with 0.1 or less recency belong to the change sets committed in the first 30 days. The rest of the graph corresponds to 90% of the time covered in the graph. Therefore, we can set the distance threshold to 10 (See the marked point in Fig. 3.1), which allows us to use all direct relations from the last 90% of the days in the graph.

$$Distance = \frac{1}{Recency} = \frac{1}{1 - \frac{270}{300}} = \frac{1}{\frac{30}{300}} = \frac{1}{0.1} = 10$$

Also, 10 seems like a good trade-off point as you see in the plot of distance against recency (Fig. 3.1). For example, a distance of 2 would not be useful because even the most recent commits have a distance of 1 on their edges. Also, if we use a larger distance limit like 100, nearly everybody could reach almost every file. Thus, we continue with 10. After this point, we continue with 10 as the distance threshold unless otherwise stated.

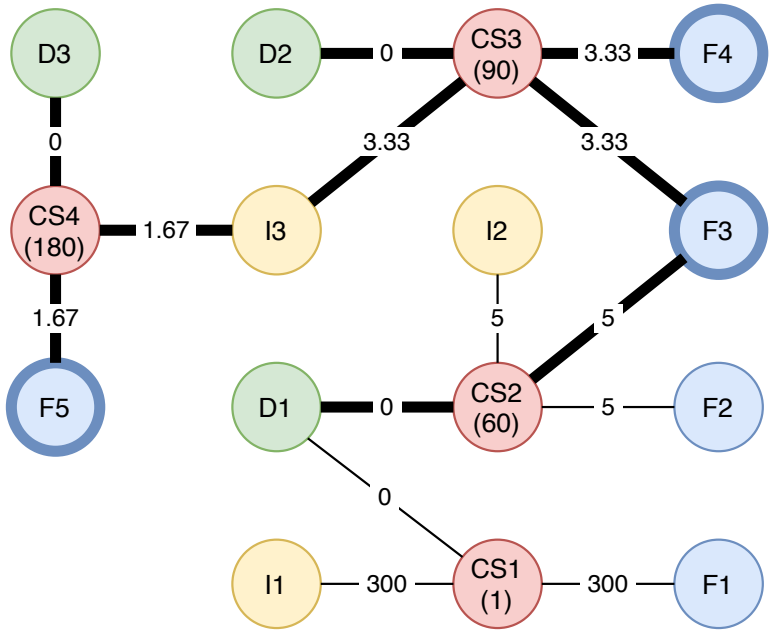


Figure 3.3: Visited edges and reachable files are highlighted to illustrate how the reachable files by D2 are found. (D: Developer, F: File, CS: Change Set, I: Issue)

Algorithm 1 Finding Reachable Files

```
1: function DevToFiles(graph, threshold)
2:     devs ← GetDevelopers(graph)                          ▷ list
3:     devToReachableFiles ← HashMap()                      ▷ string to list
4:     for dev in devs do
5:         reachableFiles ← DFS(graph, dev, threshold)
6:         devToReachableFiles.put(dev, reachableFiles)
7:     return devToReachableFiles
```

Fig. 3.3 shows how reachable files for D2 are found in the sample graph. The highlighted files (F3, F4, F5) are reachable by D2. While finding these reachable files, we run a depth-first search (DFS) algorithm starting from D2 with a stopping condition for reaching the distance threshold. The highlighted edges show the visited edges when DFS is started from the D2 node. Also, the DFS algorithm does not go through another developer node. For example, the algorithm stopped when it encountered the node of D1. Algorithm 1 shows the pseudo code for finding reachable files for each developer.

Table 3.1: Reachable files and file coverage of each developer in the sample artifact traceability graph (Assuming there are five files in the project)

Developer	Reachable Files	File Coverage
D1	F2 and F3	40%
D2	F3, F4 and F5	60%
D3	F3, F4 and F5	60%

Algorithm 2 Finding Jacks

```
1: function FindJacks(graph)
2:     devToFiles ← DevToFiles(graph, threshold)          ▷ Algorithm 1
3:     devToFileCoverage ← HashMap()                      ▷ string to float
4:     numFiles ← GetNumFiles(graph)
5:     for dev in devToFiles.keys() do
6:         numDevFiles ← devToFiles.get(dev).length()
7:         fileCoverage ← numDevFiles / numFiles
8:         devToCoverage.put(dev, fileCoverage)
9:     return SortByValue(devToCoverage)
```

3.2.2 Identifying Jacks

While finding jacks, we sort the developers in descending order according to their file coverage in the software project. *File coverage* is simply the ratio of the number of reachable files by the developer to the number of all files in the project, not just currently available files in the graph. Equation 3.3 shows the file coverage of some developer d.

$$File\ Coverage_d = \frac{\#\text{ of reachable files by } d}{\#\text{ of all files in the project}} \tag{3.3}$$

Table 3.1 shows the reachable files and file coverages for each developer in the sample artifact traceability graph given in Fig. 3.2. Algorithm 2 shows the pseudo code of finding jacks. First, it finds reachable files for each developer, then calculates file coverage scores for developers. Finally, it returns developers in descending order according to their file coverage scores.

3.3 Mavens (RQ 1.2)

By definition, mavens are the rare experts of specific parts, files or modules of the project. As we stated in Section 3.2, the source files in a software project are the reflection of the knowledge (i.e., know-how). Since mavens are the rare expert developers on specific parts, they have knowledge that the others do not have. Thus, we need to find lesser-known parts of the project.

3.3.1 Rarely Reached Files

First, reaching a file through the edges in the artifact graph means knowing the file. To meet the maven definition, we can use the files only reached by a limited number of developers. We call such files *rarely reached files*, and we set this limit to 1, which means that the files reached by only one developer are the *rarely reached files*. This could be a configurable parameter according to the size of the project. For example, for the graph given in Fig. 3.2, F2 is a *rarely reached file*. Actually it is the only one as it can be seen in Table 3.1.

Algorithm 3 shows how to find rarely reached files. It is assumed that $devToRareFiles$ is initialized with developer names and empty lists. Also, $InvertMapping$ function generates a mapping from values to keys. For instance, it inverts the hashmap $\{D1 : [F1], D2 : [F1, F2]\}$ to the hashmap $\{F1 : [D1, D2], F2 : [D2]\}$.

Algorithm 3 Finding Rarely Reached Files

```
1: function DevToRareFiles(graph, threshold)
2:     devToFiles ← DevToFiles(graph, threshold)          ▷ Algorithm 1
3:     fileToDevs ← InvertMapping(devToFiles)
4:     devToRareFiles ← HashMap()                         ▷ string to list
5:     for file in fileToDevs.keys() do
6:         devs ← fileToDevs.get(file)
7:         if devs.length() is 1 then
8:             devToRareFiles.get(devs.get(0)).append(file)
9:     return devToRareFiles
```

Algorithm 4 Finding Mavens

```
1: function FindMavens(graph, threshold)
2:     devToRareFiles ← DevToRareFiles(graph, threshold)   ▷ Algorithm 3
3:     devToMavenness ← HashMap()                          ▷ string to float
4:     numRareFiles ← GetNumRareFiles(devToRareFiles)
5:     for dev in devToRareFiles.keys() do
6:         numDevFiles ← devToRareFiles.get(dev).length()
7:         mavenness ← numDevFiles / numRareFiles
8:         devToMavenness.put(dev, mavenness)
9:     return SortByValue(devToMavenness)
```

3.3.2 Identifying Mavens

To find mavens, we consider the number of the rarely reached files of the developers. For a better comparison among developers, we define mavenness of a developer d as follows:

$$Mavenness_d = \frac{\text{\# of rarely reached files of } d}{\text{\# of all rarely reached files}} \tag{3.4}$$

While finding mavens, first we find reachable files as explained in Section 3.2.1, then we find rarely reached files as explained in Section 3.3.1 and given in Algorithm 3. Finally, we calculate mavenness scores and sort the developers according to their mavenness in descending order. Algorithm 4 shows the procedure.

3.4 Connectors (RQ 1.3)

Connectors are the developers who are involved in different sub-projects or teams. The main idea behind the connector definition is connecting developers who have no other connections, in other words, being the bridge between different groups of developers. Using node centrality, we identify this type of developers on artifact traceability graphs defined in Section 3.1.

3.4.1 Calculating Betweenness Centrality:

Betweenness centrality of a node is based on the number of shortest paths passing through that node. Freeman [37] discussed that betweenness centrality is related to control of communication. Also, Bird et al. [38] used betweenness centrality to find the gatekeepers in the social networks of mail correspondents. Therefore, we hypothesize that betweenness centrality can be a measure to find connectors. Betweenness centrality of some node v where V is the set of nodes, s and t are some nodes other than v:

$$c_B(v) = \sum_{s \neq v \neq t \epsilon V} \frac{\text{\# of shortest paths passing through } v}{\text{\# of shortest } (s,t)\text{-paths}} \tag{3.5}$$

For a better comparison among developers, betweenness values are normalized with $2/((n-1)(n-2))$ where n is the number of nodes in the graph. For betweenness centrality related operations, we use NetworkX package [39], which uses faster betweenness centrality algorithm of Brandes [40].

To use betweenness centrality, we need a graph composed of only developers because we are looking for developers who connect other developers to each other. Sulun et al.[41] proposed a metric, *know-about*, to find how much developers know the files. They found different paths between the files and the developers in the artifact graph and defined *know-about* as the summation of the reciprocals of the path lengths. Similarly, we propose to use different paths between developers to find how much they are connected in the artifact graph. The next section explains the details.

3.4.2 Constructing the Developer Graph

The developer graph is a projection of the artifact traceability graph. It defines distances directly between developers in a different way, not as we mentioned in Section 3.1. When projecting an artifact graph to a developer graph, we

find all different simple paths (paths that do not have repeating nodes) between each developer pair with a depth limit of 4. Since connector definition is not about knowing the files but about connecting the other developers, recency is not a concern and it is assumed that all edges in the artifact graph have the same distance of 1. Thus, the depth limit of 4 means that the maximum path length can be 4. Therefore, in the traceability graph, two developers can be connected through the paths with a length of 2 (through a *change set* node), or the paths with a length of 4 (through two different *change set* nodes and a *file* node connected to them). These kinds of paths can be seen in the sample graph in Fig. 3.4. We find the paths between two developers through software artifacts, not through other developers. For example, in Fig. 3.4, there is a path between D2 and D3 through D1, but we interpret this path as the combination of two paths: D1-D2 and D1-D3. Since the developers in the same team potentially work on the same group of files and these files will be close to each other (they will be connected through change sets because they will be changed by the same group frequently) in the traceability graph, the method mentioned above finds the paths between the developers in the same group. So, the developers who have connections in different groups will be favored in betweenness centrality calculations.

After finding the paths between each developer pair, we define a new distance metric, Reciprocal of Sum of Reciprocal Distances (RSRD). We define RSRD as follows when D denotes the set of all distances between two developers (i.e., the set of lengths of all different paths between two developers) and d is a distance in D:

$$RSRD = \Big[\sum_{d \epsilon D} d^{-1}\Big]^{-1} \tag{3.6}$$

Reciprocals of distances make larger contributions to the score for closer nodes. For example, $\frac{1}{2} > \frac{1}{4}$ and the path with length of 2 makes a larger contribution. After summing the contributions of all reciprocal distances, larger values represent a stronger connection. For example, $\frac{1}{2} + \frac{1}{4} = \frac{3}{4}$ means a stronger connection than

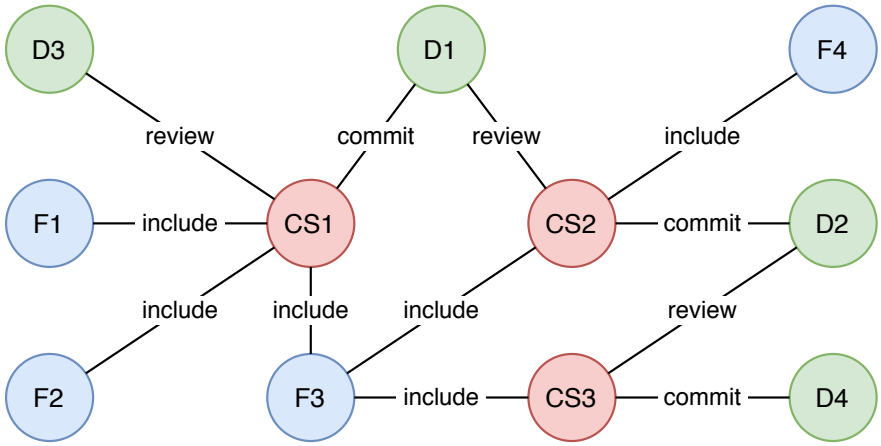


Figure 3.4: Another sample artifact traceability graph. (D: Developer, F: File, CS: Change Set)

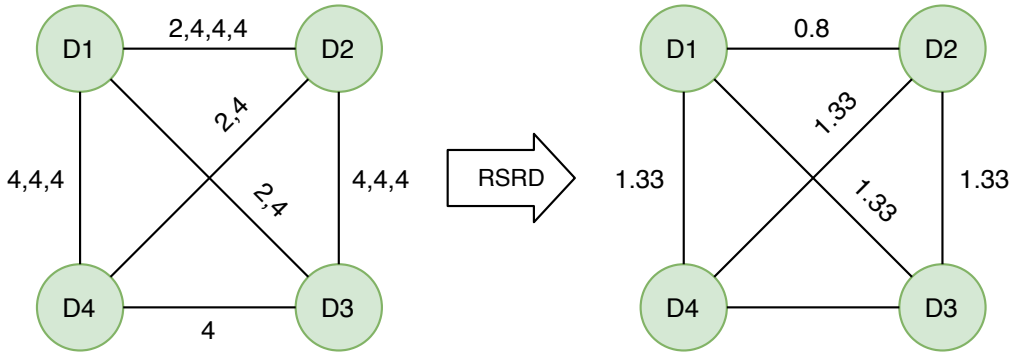


Figure 3.5: Sample developer graph. (D: Developer)

$\frac{1}{4} + \frac{1}{4} = \frac{2}{4}$. To use betweenness centrality, we need to inverse the result of this summation, because the nodes with stronger connections need to be closer. For example, for the numbers in the previous example, $\frac{4}{3} = 1.33$ is smaller than $\frac{4}{2} = 2$, and it means a closer relation. At the end, a smaller RSRD score represents a closer relationship between two developers, just like any other distance metric.

Fig. 3.5 shows how the developer graph is constructed from the sample artifact graph in Fig. 3.4. For example, (2, 4, 4, 4) are the distances of the different paths between D1 and D2 in Fig. 3.4, and the RSRD between these two developers is calculated as follows:

$$(2^{-1} + 4^{-1} + 4^{-1} + 4^{-1})^{-1} = \left[\frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}\right]^{-1} = \left[\frac{5}{4}\right]^{-1} = 0.8$$

Algorithm 5 Calculating RSRD

```
1: function CalculateRsrd(graph, maxDepth)
2:     devs ← CurrentDevelopers(graph)                          ▷ list
3:     targetDevs ← devs          ▷ copy of devs to keep target developers
4:     devPairToPaths ← HashMap()                    ▷ string pair to list
5:     for startDev in devs do
6:         targetDevs ← targetDevs − startDev
7:         paths ← DFS(graph, startDev, targetDevs, maxDepth)
8:         for path in paths do
9:             endDev ← path.getLast()
10:            devPairToPaths.get((startDev, endDev)).append(path)
11:    devPairToRsrd ← HashMap()                     ▷ string pair to float
12:    for devPair in devPairToPaths.keys() do
13:        srd ← 0
14:        paths ← devPairToPaths.get(devPair)
15:        for path in paths do
16:            srd ← srd + 1/path.length()
17:        rsrd ← 1/srd
18:        devPairToRsrd.put(devPair, rsrd)
19:    return devPairToRsrd
```

Algorithm 5 shows the pseudo code for calculating RSRD values for a given graph and a depth limit. First, it runs a DFS algorithm starting from each developer node to find the paths to the nodes of the target developers. Then, for each developer pair, it calculates an RSRD value by using the path lengths.

3.4.3 Identifying Connectors

When identifying connectors, we use the betweenness centrality of developers in the developer graph. Algorithm 6 shows the procedure. First, it finds different paths and RSRD values for each developer pair as mentioned above. Then, it creates a developer graph with these RSRD values and finds betweenness centrality for each developer in that graph. Finally, it sorts developers in descending order according to their centrality.

Algorithm 6 Finding Connectors

```
1: function FindConnectors(graph)
2:     devPairToRsrd ← CalculateRsrd(graph)
3:     devGraph ← DeveloperGraph(devPairToRsrd)
4:     devToBtwn ← BetweennessCentrality(devGraph)
5:     return SortByValue(devToBtwn)
```

3.5 Identifying the Significant Set of Key Developers

We return a sorted list of developers for each type of key developer. We sort developers according to file coverage (proportional to reachable files) for *jacks*, mavenness (proportional to rarely reached files) for *mavens* and betweenness centrality in the developer graph for *connectors*. In these sorted lists, the developers near the top are more significant than the others, thus we can set a threshold and call the developers above the threshold *the significant set of developers*. To identify them, we find the developers who made at least 80% of the contributions (similar to the Pareto principle) for each category separately. For *jacks*, we find the developers who can reach at least 80% of the reachable files. For *mavens*, we identify the developers who can reach at least 80% of the rarely reached files. For *connectors*, we identify the developers who are directly connected to at least 80% of the other developers in the developer graph.

3.6 Replacement for Leaving Developers (RQ 2)

Developers leave and join projects due to various reasons like changing the team in the same company or starting to work in another company. Some level of developer turnover happens all the time. It is a risk for the software projects [4] and inevitable in practice. When a developer leaves the project, other developers have to take over the jobs of that developer. By using the reachable files explained in Section 3.2.1, we propose to recommend developers who know the files known by the leaving (i.e., former) developer. For a given developer in the artifact

Algorithm 7 Finding Replacements

```
1: function FindReplacement(leavingDev, threshold)
2:     devToFiles ← DevToFiles(graph, threshold)                ▷ Algorithm 1
3:     leavingDevFiles ← devToFiles.get(leavingDev)
4:     allDevs ← devToFiles.keys()
5:     otherDevs ← allDevs − leavingDev
6:     devToOverlappingKnowledge ← HashMap()                    ▷ string to integer
7:     for dev in otherDevs do
8:         devFiles ← devToFiles.get(dev)
9:         intersection ← Intersection(devFiles, leavingDevFiles)
10:        overlappingKnowledge ← intersection.length() / leavingDevFiles.length()
11:        devToOverlappingKnowledge.put(dev, overlappingKnowledge)
12:    return SortByValue(devToOverlappingKnowledge)
```

graph, we recommend a list of developers sorted by the intersection percentage of the reachable files of two developers, the former one and the recommended one. In other words, we use overlapping knowledge amounts while recommending replacements. The following equation shows the overlapping knowledge of a recommended developer (rd) and a leaving developer (ld) when F represents the set of reachable files by the developer.

$$OverlappingKnowledge(rd, ld) = \frac{|F_{rd} \cap F_{ld}|}{|F_{ld}|} \tag{3.7}$$

For example, overlapping knowledge is 0.33 when rd reaches [F1, F2] and ld reaches [F2, F3, F4]. Algorithm 7 shows the pseudo code of our replacement recommendation algorithm.

3.7 Knowledge Distribution: Balanced or Hero (RQ 3)

While identifying *jacks* in the projects, we found which developers reach which files, then the file coverage ratio of each developer. In a balanced team, it is expected that the developers' file coverage ratios should be close enough, and they should not fluctuate much. In different studies [34, 35, 31], imbalance knowledge distribution is found in software projects. Also, Agrawal et al. [31] called the imbalanced projects *hero projects* and called the others *non-hero projects*. In our case, we use a similar terminology, *hero* and *balanced* teams (or projects).

Even though all developers work on the same project, some contribute more and some contribute less. Also, particularly in OSS projects, contribution frequency affects the knowledge since a group of developers does not contribute continuously every day or every week. Therefore, the contribution amount differs among developers. We assume that the knowledge distribution in a balanced team should follow a normal distribution. The file coverage score given in Equation 3.3 (Section 3.2.2) is to measure the knowledge of the developers. So, we assume that the file coverage distribution should follow a normal distribution in a balanced team. To decide whether the file coverage distribution is normal or not, we propose to use statistical tests. The statistical tests for normality are to decide if the samples come from a normally distributed population or not. In our case, file coverage values are not the samples from a population but they somehow represent the knowledge distribution. Therefore, we claim that using a normality test could decide whether a development team has a balanced knowledge distribution or not.

Shapiro-Wilk test [42] is commonly used for testing normality and suitable for smaller numbers of samples. Suitability for small sample counts is important in our case since software projects are generally developed by tens of active developers. Also, Razali and Wah [43] inspected the power of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling normality tests. Then, they concluded that the Shapiro-Wilk test is the most powerful normality test. Thus, we use

Algorithm 8 Finding if Balanced or Hero Team (or Project)

```
1: function BalancedOrHero(graph)
2:     devToCoverage ← FindJacks(graph)                    ▷ Algorithm 2
3:     if devToCoverage.length() < 3 then
4:         return null
5:
6:     p ← ShapiroWilkTest(devToCoverage.values())
7:     if p > 0.05 then
8:         return "balanced"
9:
10:    return "hero"
```

the Shapiro-Wilk normality test in our approach with the standard 0.05 alpha value. It tests the null hypothesis(H_0) which claims that the samples come from a normal distribution. If the statistical test rejects H_0, the distribution is statistically significantly different from a normal distribution, and we label that team as *hero*. If the test cannot reject H_0, the samples are not statistically significantly different from a normal distribution (i.e., they can be normally distributed), and we label that team as *balanced*. Not rejecting H_0 does not prove the samples are normally distributed. It means the test cannot reject H_0 with the available samples. Therefore, if we cannot label a team as *hero* team, we call them *balanced*. A corner case of this approach is that the number of developers in a team can be less than three in small-scale projects. In that case, performing a test is not possible, and we do not label the team at all. Algorithm 8 shows the pseudo code for finding whether the team is balanced or not.

Chapter 4

Dataset

4.1 Selecting Datasets

As we mentioned before, we use software artifacts from project history to construct the artifact traceability graph. More specifically, our approach needs change sets (i.e., commits) and their related data such as author, changed files and linked issues. Rath and Mader [1] published datasets for 33 OSS projects, SEOSS 33. All 33 datasets are available online.[1] Out of 33 projects, we selected Apache Hadoop[2], Apache Hive[3], Apache Pig[4], Apache HBase[5], Apache Derby[6] and Apache Zookeeper[7] since these six projects have the highest issue and change set link ratios among SEOSS 33 datasets. The datasets include data from version control systems (e.g., Git) and issue tracking systems (e.g., Jira). Table 4.1 shows the details for each dataset with a varying number of issues and change sets.

[1] https://bit.ly/2wukCHc (Accessed on 24 Sep 2020)
[2] https://hadoop.apache.org/ (Accessed on 24 Sep 2020)
[3] https://hive.apache.org/ (Accessed on 24 Sep 2020)
[4] https://pig.apache.org/ (Accessed on 24 Sep 2020)
[5] https://hbase.apache.org/ (Accessed on 24 Sep 2020)
[6] http://db.apache.org/derby/ (Accessed on 24 Sep 2020)
[7] https://zookeeper.apache.org/ (Accessed on 24 Sep 2020)

Table 4.1: Dataset details before preprocessing [1]

Project	# of Developers	Time Period (months)	# of Issues	# of Change Sets	Change Sets Linked to Any Issue (%)
Hadoop	216	150	39,086	27,776	97.13
Hive	222	113	18,025	11,179	96.34
Pig	29	123	5,234	3,134	92.85
Hbase	266	131	19,247	14,331	90.06
Derby	37	160	6,969	8,156	83.17
Zookeeper	69	116	2,907	1600	87.12

4.2 Preprocessing

Data is already extracted from the version control and issue tracking platforms and provided in an SQL dataset. Nonetheless, we processed the data in order to prevent errors and calculate specific fields. We did not use all the information in the datasets; *change_set*, *code_change* and *change_set_link* tables were enough to create nodes and edges for developers, changes sets, issues, files and relations among them.

We processed change sets from the *change_set* table ordered by *commit_date*, extracted the data required and dumped them into a file as a JSON formatted string of change sets in the temporal order. For each change set, we extracted the following information: commit hash, author, date (*commit_date*), issues linked, set of file paths with their change types, number of files in the project (after the change set).

In the data extraction, the following points are important:

- We only extracted the code changes in java files which, we assumed, end with ".java" extension. If a change set has no code change including a java file, we ignored it completely.
- We ignored the merge change sets (*is_merge* is 1) since they could inflate the contributions of some developers [31].

- We created a look-up table for each project to detect different author names of the same authors. They are created manually by looking at the developer names and their email addresses. For example, "John Doe" and "Doe John" can be the same developer if they share a common email address. We used this table to correct the author names by replacing them.

- In order to calculate file coverage score (See Section 3.2.2), we needed the number of files in the project after each change set. Thus, we tracked the set of current files over time. After each code change, we removed the file if its change type is *DELETE*, and we added the file if its change type is *ADD*.

- Git does not track *RENAME* situations explicitly. When a file is renamed, it is a *DELETE* and an *ADD* for Git (if there is no change in the file)[8]. In the *code_change* table, there are three *change_type*s: *ADD*, *DELETE* and *MODIFY*. In that case, we needed to handle renames because it would affect our traceability graph and change the knowledge balance among developers. We treated (*DELETE*, *ADD*) pairs in the same change set (commit) as *RENAME* when the following conditions were satisfied:

  - Both have the same file name (file paths are different).
  - The number of lines deleted in *DELETE* code change and the number of lines added in *ADD* code change are equal.

  So, our rename algorithm only detects file path changes and does not check file contents. For example, it detects *RENAME* when "example.java" moved from "module1/example.java" to "module2/example.java" but does not detect it when the file content is changed. Since we used the datasets from SQL tables [1] directly and we did not mine them from Git repositories ourselves, we used the heuristic given above.

- In the Hadoop dataset, we detected that there were duplicate commits. Even though the commit hashes were different, the rest of the extracted

[8]https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository (Accessed on 24 Sep 2020)

Table 4.2: Dataset details after preprocessing

Project	# of Developers	# of Change Sets	# of Change Sets added or modified files more than 10	# of Change Sets added or modified files more than 50
Hadoop	153	15,178	1,900 (12.52%)	129 (0.85%)
Hive	180	9,030	1,062 (11.76%)	127 (1.41%)
Pig	25	2,401	240 (10.00%)	32 (1.33%)
Hbase	197	10,963	1,314 (11.99%)	155 (1.41%)
Derby	34	6,831	475 (6.95)	65 (0.95)
Zookeeper	47	930	116 (12.47)	10 (1.08)

data were identical. This situation only applies to Hadoop, the same preprocessing steps did not produce such a situation for Hive and Pig. We removed these change sets by using string comparison for all parts of the JSON string except the *commit hash*.

Table 4.2 shows the number of change sets for each dataset after preprocessing. Also, we share our implementation online (See Appendix B), and it includes the preprocessing script.

4.3 Handling Large Change Sets

Change set means a set of file changes, and it is called a large change set when the number of changed files is more than a specific number. For example, initial commits of a project most probably include many files, and it is a typical example for large commits. Another example is moving a project into another project. In that case, its change set includes all the files in the added project.

Committing a large number of files in one change set is not considered to be a good practice in software engineering. Sadowski et al.[44] claimed that 90% of the changes in Google modify less than 10 files. Also, Rigby and Bird [45] excluded the changes that contain more than 10 files in their case studies. In our experiments, we used a looser limit for excluding change sets. In the following, we explain the details on removing the large change sets:

- Regardless of the size of a change set, we applied changes to the traceability graph for *DELETE* and *RENAME* types since the knowledge of deleted files is not required after that point and renamed files need to proceed with their new names.

- If a change set includes more than 50 files which are added or modified, we ignored these *ADD* and *MODIFY* code changes. We did not use 10 as the limit because we did not want to lose 6.95-12.52% of the datasets (See Table 4.2). Also, sometimes large commits can exist even though it is not a good practice. Our purpose is to handle the initial commits of the projects and project movements. So, choosing 50 is a good trade-off for the limit of the number of files added or modified in a change set. It is neither small to cause losing 6.95-12.52% of the datasets nor large to include commits like initial commits.

Because we needed to keep track of *DELETE* and *RENAME* types, we did not exclude large change sets in preprocessing. Instead, we detected large change sets while adding change sets to the graph, then we applied changes according to the rules given above.

Chapter 5

Case Studies

In this section, we share the details of our evaluation setup and the results for each RQ.

5.1 Evaluation Setup

In the experiments, we used the NetworkX package [39] for graph operations. In order to prevent potential bugs, we used its built-in functions whenever possible (e.g., calculating betweenness centrality, finding paths between developers). However, we implemented the DFS algorithm for reachable files in Algorithm 1 because it was very specific to our case (e.g., the stopping condition is different). Our source code is available online (See Appendix B).

How much time the artifact graph should cover is a parameter in our method. We chose a sliding window approach over an incremental window in time, in other words, the artifact graph always includes the change sets committed in a constant time period. The followings are the reasons behind this choice:

- If the time period of the graph changes over time, the meaning of the recency changes. For example, 0.9 recency means 30 days ago in a 300-day graph while the same recency corresponds to 50 days ago in a 500-day graph. Thus, keeping the time period (sliding window size) constant enables recency scores to have the same meaning in different time points.

- Keeping every artifact from history enlarges the graph every day, and the algorithms run slower in larger graphs. Therefore, removing unnecessary parts (the artifacts older than one year) means less run time, in other words, it is more efficient.

- In OSS projects, there is no data regarding leaving developers. If we keep every artifact from history, we should calculate scores even for former developers. Therefore, removing old artifacts enables us to keep track of the current developers. If the graph keeps the last 365 days, we assume the developers who contributed to the project in the last 365 days are the current active developers.

We used six-month (180 days) and one-year (365 days) sliding windows in our experiments. Figure 5.1 shows how the included days change in iterations. The numbers on the figure come from the Hive dataset when the sliding window is one-year. "3367 days" corresponds to the number of days after preprocessing. There are 3003 iterations including the initial window. We tracked the dates over change sets. When forwarding the window one day, first we remove one day from the beginning of the window, then we add one day to the end of the window. For

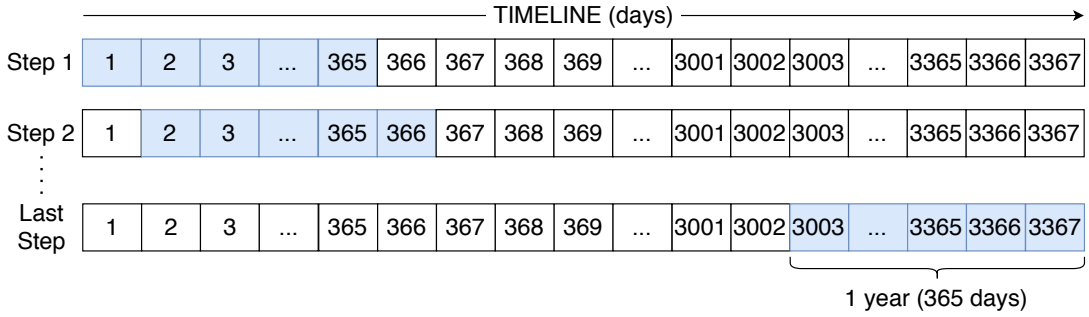


Figure 5.1: Evaluation Setup (one-year sliding window, Hive experiment)

each iteration, we calculated scores for jacks, mavens and connectors, then we reported them and their scores in descending order. Here we considered scores less than 5e-6 as 0 (zero) to make the results more readable especially in the proof of concept tool (See Section 6). The same procedure was repeated for all other projects.

5.2 Results

In this section, we share the results for each RQ.

5.2.1 Results for Identifying Key Developers (RQ 1)

Since we propose to use *jack*, *maven* and *connector* as the key developer categories for the first time and there is no classification of developers for these types in the literature, we are not able to compare our approach with others. Also, since we conducted our experiments on OSS projects, we have no data for developer labels for these projects. However, we can show that the results of our approach are compatible with other statistics of the projects.

To validate our approach, we propose to use developers' comments on issues. *Jacks* are the developers who have broad knowledge by definition, and we identified them by finding their file coverage in the project. Therefore, the *jacks* should be involved in issues such as bugs and enhancements more than other developers. We claim that, by definition, the top *jacks* and the top commenters in the project's issue tracking system (e.g., Jira) should be mostly the same developers. However, we cannot claim that *mavens* and *connectors* should be among the top commenters. To validate the results of these categories, we offer to use the developers who are *jack*, *maven* and *connector* at the same time, in other words, the intersection of all kinds of key developers. In that way, we include *mavens* and *connectors* to our validation, and we show how the intersection developers overlap with the top commenters. While using the intersection developers, we sorted

them by their jack score (i.e., file coverage) since we cannot combine betweenness centrality, mavenness score and file coverage properly. So, in the case studies, we examined the jacks and the intersection developers.

The datasets [1] we used for experiments include data from issue tracking systems (e.g., Jira). In the *change_set_link* table, there are links between issues and change sets, which means we can use the comments made to issues in the traceability graph. The datasets supply the *display_name* of the commenters in the *issue_comment* table. The names in the *display_name* field match the developer names in the *author* field of the *change_set* table. So, we can match committers with commenters and find how many comments developers made to the issues in the graph. To increase the validity of the number of comments for each developer, we corrected the commenter names by using the look-up table created manually in preprocessing (See Section 4.2).

The *Key Developers* columns in Table 5.1 show the accuracy of our approach when we treat the top commenters as the actual key developers (i.e., ground truth). "Top commenters" means a ranked list of commenters according to the number of comments that they made to the issues in the last six months or the last year depending on the sliding window size. The predicted key developers by our model are consistent with the top commenters up to 98%.

Accuracy is calculated as shown in Equation 5.1, where KD is the ranked list of key developers, C is the ranked list of commenters, D is the set of dates (i.e., days or iterations in Figure 5.1) and k refers to the numbers in *Topk* phrases in Table 5.1. For example, the accuracy of day d for (Top-3, Top-5) cell is calculated as follows: If $C_d(3) = \{D1, D2, D3\}$ and $KD_d(5) = \{D1, D2, D4, D5, D6\}$, the accuracy is $\frac{|\{D1,D2\}|}{|\{D1,D2,D3\}|} = \frac{2}{3} = 0.67$.

$$Mean\ Accuracy(k_1, k_2) = \frac{1}{|D|} \sum_{d}^{D} \frac{|C_d(k_1) \cap KD_d(k_2)|}{|C_d(k_1)|} \tag{5.1}$$

Since there is no comparable approach that finds our subcategories of key developers, we used the Monte Carlo simulation as a baseline approach. We

randomly selected the key developers for each day from the existing developers in the graph, in other words, from the developers who committed changes to the source code in the sliding window period. While producing random developers, we considered the number of key developers in our results since the simulation should provide random results in the same structure. For example, we selected four random developers if our approach found four jacks on that day even if k_2 is five. Then, we calculated mean accuracies in the same way shown in Equation 5.1. This experiment with random key developers is repeated 1000 times. The *Randomly Selected Developers* columns in Table 5.1 show the average accuracies of 1000 simulations. Also, *Average Improvement* columns show the improvement of our approach over the random selection on average.

The selected projects have different scales as seen in Table 4.1 and Table 4.2. Pig, Derby and Zookeeper are small projects with tens of developers while the others have hundreds of developers in their whole history. Even though Hadoop, Hive and HBase have hundreds of developers and their time periods are not that different (See Table 4.1). Hadoop has a lot more change sets and issues than Hive and HBase (See Table 4.1 and Table 4.2). The average number of active developers (sliding window size 1 year) for each project, in other words, the average numbers of developers in the traceability graph over time are 49.86 in Hadoop, 35.08 in Hive, 9.05 in Pig, 32.19 in HBase, 10.90 in Derby and 8.25 in Zookeeper. So, it is clear that Hadoop is a more active project than Hive and HBase. Also, the differences between the results of projects in Table 5.1 infer the same conclusion. Both the results of our algorithms and the results of the Monte Carlo simulation show that the more active developers exist, the harder it becomes to predict key developers. Even though the accuracies are different among the projects due to the fact mentioned above, our results are better than the results of the random model for all cases (See Table 5.1). Also, the key developers predicted by our approach and the top commenters overlap up to 98%.

Table 5.1: Mean accuracies (%) for the key developers found by our approach vs. the developers selected randomly in the Monte Carlo simulation. Average improvement (%) means improvement of our approach over random selection.

Key Developer Category	Projects	Top Commenters	Sliding Window Size is Six Months									Sliding Window Size is One Year								
			Key Developers				Randomly Selected Developers				Average Improvement	KeyDevelopers				Randomly Selected Developers				Average Improvement
			Top-1	Top-3	Top-5	Top-10	Top-1	Top-3	Top-5	Top-10		Top-1	Top-3	Top-5	Top-10	Top-1	Top-3	Top-5	Top-10	
JACK	HADOOP	Top-1	12.88	29.11	40.45	60.03	2.54	7.62	12.73	25.47	192.43	6.82	19.04	27.10	50.47	1.96	5.86	9.79	19.57	196.85
		Top-3	-	23.87	33.01	51.14	-	6.84	11.41	22.86		-	22.65	30.20	47.81	-	5.58	9.29	18.59	
		Top-5	-	-	27.26	45.31	-	-	11.47	22.95		-	-	29.62	47.40	-	-	9.58	19.18	
		Top-10	-	-	-	41.84	-	-	-	21.63		-	-	-	41.75	-	-	-	19.13	
	HIVE	Top-1	37.23	67.72	79.02	89.18	8.29	24.88	40.69	63.85	103.05	44.16	71.43	81.65	92.21	6.24	18.71	31.21	58.14	172.38
		Top-3	-	51.65	65.68	77.27	-	22.33	36.60	57.55		-	54.78	70.20	84.12	-	17.48	29.12	54.17	
		Top-5	-	-	54.02	68.96	-	-	32.42	51.68		-	-	57.02	73.28	-	-	26.61	49.54	
		Top-10	-	-	-	48.88	-	-	-	36.86		-	-	-	55.31	-	-	-	38.85	
	PIG	Top-1	57.55	85.87	93.95	95.54	15.28	45.85	69.65	92.48	53.88	59.16	86.23	88.90	89.91	11.81	35.37	56.37	83.57	84.53
		Top-3	-	70.98	81.60	85.61	-	41.28	62.36	82.61		-	75.27	85.26	89.99	-	34.96	55.63	83.07	
		Top-5	-	-	66.35	74.95	-	-	52.61	72.97		-	-	66.91	79.66	-	-	46.60	73.74	
		Top-10	-	-	-	51.69	-	-	-	50.69		-	-	-	59.15	-	-	-	55.84	
	HBASE	Top-1	54.67	80.38	87.08	93.01	8.39	25.20	39.55	58.18	144.69	59.50	88.81	94.83	98.64	6.92	20.74	34.45	55.82	204.60
		Top-3	-	58.45	68.59	74.99	-	21.81	33.91	47.96		-	62.80	74.69	80.00	-	17.79	29.54	46.60	
		Top-5	-	-	52.56	60.62	-	-	28.37	40.72		-	-	58.76	66.88	-	-	26.56	42.05	
		Top-10	-	-	-	45.60	-	-	-	33.12		-	-	-	50.91	-	-	-	34.54	
	DERBY	Top-1	26.17	40.28	43.03	45.18	7.38	18.16	24.55	37.59	70.16	11.89	27.91	30.37	30.97	3.31	9.49	14.25	21.84	94.64
		Top-3	-	42.33	50.03	53.57	-	21.78	31.03	48.06		-	46.43	54.57	57.81	-	19.15	29.00	48.56	
		Top-5	-	-	39.89	46.35	-	-	26.74	42.55		-	-	43.58	51.80	-	-	26.60	45.02	
		Top-10	-	-	-	38.70	-	-	-	36.91		-	-	-	42.47	-	-	-	38.90	
	ZOOKEEPER	Top-1	39.03	62.80	72.70	75.60	13.58	40.73	62.82	70.41	30.51	39.28	65.90	77.35	83.10	12.75	38.29	62.01	77.10	39.08
		Top-3	-	47.29	65.11	69.08	-	39.09	60.11	67.02		-	48.59	68.28	75.48	-	35.85	58.01	71.41	
		Top-5	-	-	59.23	63.46	-	-	55.80	62.56		-	-	62.34	71.57	-	-	55.23	68.89	
		Top-10	-	-	-	43.14	-	-	-	42.72		-	-	-	50.18	-	-	-	49.34	
INTERSECTION OF ALL (Sorted By Jack Score)	HADOOP	Top-1	13.67	34.40	48.68	64.95	2.55	7.65	12.64	21.30	247.43	7.22	21.33	34.97	59.04	1.95	5.85	9.75	17.46	238.68
		Top-3	-	27.42	38.67	54.47	-	6.86	11.32	19.15		-	23.80	32.68	52.05	-	5.57	9.28	16.65	
		Top-5	-	-	32.90	48.68	-	-	11.38	19.32		-	-	31.26	49.08	-	-	9.58	17.14	
		Top-10	-	-	-	43.82	-	-	-	18.29		-	-	-	44.38	-	-	-	17.03	
	HIVE	Top-1	55.80	75.82	83.53	89.93	8.26	14.46	16.37	19.06	300.92	56.94	75.92	81.15	87.98	6.27	10.95	12.52	14.69	423.58
		Top-3	-	45.55	51.05	57.56	-	13.02	14.76	17.30		-	49.40	55.17	61.33	-	10.39	11.92	14.02	
		Top-5	-	-	37.76	44.08	-	-	13.40	15.75		-	-	38.86	44.94	-	-	10.81	12.70	
		Top-10	-	-	-	29.84	-	-	-	12.61		-	-	-	30.48	-	-	-	10.70	
	PIG	Top-1	59.65	78.81	79.66	79.66	14.90	27.83	28.84	28.84	131.36	66.26	86.41	86.63	86.63	11.81	22.36	23.58	23.58	191.85
		Top-3	-	51.57	52.47	52.47	-	24.51	25.39	25.39		-	55.54	56.30	56.30	-	22.46	23.79	23.79	
		Top-5	-	-	37.22	37.22	-	-	22.48	22.48		-	-	39.39	39.39	-	-	20.73	20.73	
		Top-10	-	-	-	19.53	-	-	-	15.62		-	-	-	21.29	-	-	-	15.45	
	HBASE	Top-1	59.55	83.16	89.53	93.47	8.34	12.76	14.53	16.89	364.55	65.86	90.14	35.26	97.19	6.92	11.23	12.51	14.62	466.60
		Top-3	-	45.55	51.20	54.25	-	10.89	12.39	14.27		-	48.45	53.09	54.68	-	9.50	10.60	12.35	
		Top-5	-	-	35.09	37.42	-	-	10.23	11.81		-	-	36.84	38.32	-	-	9.26	10.74	
		Top-10	-	-	-	23.60	-	-	-	9.80		-	-	-	24.28	-	-	-	9.13	
	DERBY	Top-1	20.67	33.01	33.63	33.63	3.16	8.59	11.09	11.31	185.22	11.89	24.41	24.84	24.84	2.03	5.43	7.01	7.24	203.64
		Top-3	-	31.34	33.94	33.96	-	12.49	14.85	15.04		-	37.79	40.79	40.82	-	13.10	15.90	16.21	
		Top-5	-	-	25.85	25.91	-	-	13.56	13.71		-	-	29.02	29.18	-	-	15.27	15.56	
		Top-10	-	-	-	16.04	-	-	-	12.44		-	-	-	17.74	-	-	-	13.54	
	ZOOKEEPER	Top-1	42.59	49.47	49.61	49.61	12.88	18.28	18.54	18.61	103.95	44.97	55.24	55.78	55.78	12.63	19.18	19.49	19.56	124.83
		Top-3	-	27.49	27.96	28.03	-	16.53	16.70	16.75		-	34.76	35.09	35.33	-	17.49	17.73	17.78	
		Top-5	-	-	21.59	21.63	-	-	15.19	15.23		-	-	26.31	26.58	-	-	17.05	17.11	
		Top-10	-	-	-	12.53	-	-	-	10.40		-	-	-	15.40	-	-	-	12.44	

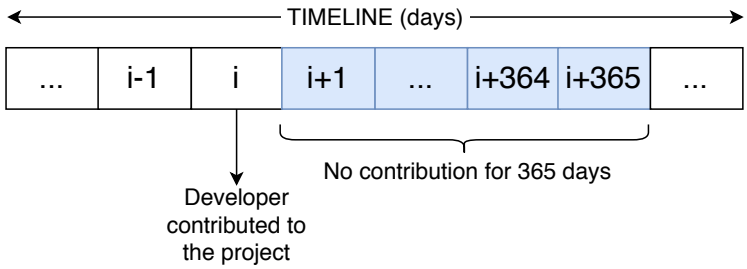


Figure 5.2: How to detect leaving developers for one-year absence limit

5.2.2 Results for Developer Replacement (RQ 2)

In a company, the developers who left or joined the projects are known by the other developers or the manager of the development team. Thus, they can use our algorithm to see who can take over the jobs of leaving developers. However, the OSS projects generally do not record who joins or leaves the team. Even if they keep such a record, it is not included in the git repositories and mining them is mostly a manual job. We prefer to use an automated approach to detect developer turnover in OSS projects with a simple condition. If a developer does not contribute to the project for a period of time (e.g., for one year), we consider that developer as a *leaving developer*.

To detect the developers who left the project, we benefit from the sliding window experimental setup explained in Section 5.1. Figure 5.2 illustrates the details of detecting developer turnover. If a developer contributes to the project in the i^{th} day and does not contribute for the following time period (one-year in Figure 5.2), we consider that the developer left the project. Therefore, we detect leaving developers when they leave the sliding window. In other words, if developers do not contribute to the project for a period of time, which we call *absence limit*, they are considered as *leaving developers*. Developers can have a break on a project or go on a holiday for a short time. To avoid misinterpretation of such situations, we do not consider a short time for absence limit, and we use six months (180 days) and one year (365 days) in our experiments. Table 5.2 shows the number of leaving developers in the whole project history for each project.

Table 5.2: Number of leaving developers

Absence Limit	Project					
	Hadoop	**Hive**	**Pig**	**HBase**	**Derby**	**Zookeeper**
Six Months	160	147	32	169	50	42
One Year	89	106	25	105	37	17

Rigby et al. [11] defined a *line* as abandoned when its developer found by *git blame* left the project, and they defined a *file* as abandoned when 90% or more of its lines are abandoned. They suggested successors (replacements) to abandoned files. Then, they considered the suggested developers to be correct if they modify the abandoned files in the future. Afterward, they reported the accuracy of their suggested successors for abandoned files, and they compared the results with the results of randomly suggested developers. We follow a similar and more explicitly defined approach in our experiments. Differently, we limit the number of checked days after a developer leaves.

In our approach, we suggest replacements by using the files reached by the leaving developer (See Section 3.6). Then, we offer to validate our approach by inspecting if the recommended replacements modify the files of the leaving developers after they leave. As shown in Figure 5.3, we limit the inspection with a time period, which is 30 days (one month) in the figure. Also, we only consider the leaving developer's files which are not reached by the recommended replacements when the developer leaves. In other words, we only consider the files that the leaving developer can reach but the recommended developers cannot reach on i^{th} day. For example, if a leaving developer can reach {*F1, F2, F3, F4*} and a suggested replacement can reach {*F1,F2*} on i^{th} day, we check the days

Figure 5.3: Validation setup of the recommended replacements for checking the next 30 days

from *(i+1)*th to *(i+30)*th. If the suggested developer modifies the *F3* or *F4* in these days, we consider our suggestion to be correct, otherwise incorrect. There are two important points here. First, we only consider MODIFY type of changes, not *ADD*, *DELETE* or *RENAME* (see Section 4.2) since adding an existing file is not possible and deleting or renaming a file does not mean a modification of the file content. Second, this approach favors the randomly selected developers because their files will not intersect much with the files of the leaving developer. Thus, the number of files to be checked will be more for randomly selected developers. Our suggestion works in a way that is proportional to the intersection of the files of the leaving developer and the files of the suggested developer. So, the number of files to be checked will be less. For the previous example, if a randomly selected developer can reach {*F3*}, we check if (s)he modifies *F1*, *F2* or *F4* in the future. Therefore, we obviously favor randomly selected developers in our evaluation setup.

We use *Top-k Accuracy* to evaluate our approach. *Top-k Accuracy* is calculated as follows where *isCorrect(r)* returns 1 if any *Top-k* recommended replacement modifies the target files in the next days, otherwise returns 0.

$$\textit{Top-k Accuracy} = \frac{1}{|R|} \sum_{r \in R} isCorrect(r) \tag{5.2}$$

We also evaluated our approach using *mean reciprocal rank (MRR)*. *MRR* is calculated as follows where *rank(r)* returns the rank of the first correct replacement in the ranked list of recommended replacements, and if none of them is correct, it returns ∞ to make the reciprocal zero (0). We used only the *Top-3* recommended replacements in our experiments. Reciprocal ranks can be 1, $\frac{1}{2}$, $\frac{1}{3}$ or 0.

$$MRR(R) = \frac{1}{|R|} \sum_{r \in R} \frac{1}{rank(r)} \tag{5.3}$$

Similar to Rigby et al. [11], as a baseline approach, we performed Monte

Table 5.3: Replacement accuracy (in percent) and MRR (in percent) when absence limit is six months (Topk phrases refer to accuracy)

Method	Project	Check next 7 days				Check next 30 days				Check next 90 days			
		Top1	Top2	Top3	MRR	Top1	Top2	Top3	MRR	Top1	Top2	Top3	MRR
Our Approach	Hadoop	13.42	21.48	30.87	20.58	28.19	45.64	56.38	40.49	43.62	59.73	69.80	55.03
	Hive	9.68	15.32	25.00	15.73	19.35	40.32	51.61	33.60	36.29	56.45	66.13	49.60
	Pig	11.54	15.38	19.23	14.74	26.92	65.38	73.08	48.72	46.15	80.77	80.77	63.46
	HBase	11.11	21.57	30.72	19.39	24.84	46.41	62.75	41.07	39.22	65.36	79.08	56.86
	Derby	9.09	13.64	20.45	13.64	27.27	45.45	47.73	37.12	43.18	68.18	75.00	57.95
	Zookeeper	3.33	13.33	13.33	8.33	6.67	16.67	30.00	16.11	10.00	30.00	43.33	24.44
Random Selection	Hadoop	5.67	10.83	15.23	9.72	15.24	27.05	36.69	24.36	26.34	44.01	56.15	39.22
	Hive	7.39	14.36	20.39	12.88	19.95	35.42	46.73	31.45	35.20	55.50	67.93	49.49
	Pig	6.50	13.19	18.77	11.71	25.19	45.50	61.23	40.59	40.77	65.38	78.65	57.50
	HBase	7.88	14.71	20.50	13.22	21.24	36.22	47.16	32.28	35.16	55.28	67.44	49.27
	Derby	8.52	15.59	21.77	14.12	23.82	40.16	51.11	35.64	42.50	62.93	71.70	55.64
	Zookeeper	7.57	14.67	19.93	12.87	14.13	24.40	32.20	21.87	25.37	42.73	54.23	37.88

Table 5.4: Replacement accuracy (in percent) and MRR (in percent) when absence limit is one year (Topk phrases refer to accuracy)

Method	Project	Check next 7 days				Check next 30 days				Check next 90 days			
		Top1	Top2	Top3	MRR	Top1	Top2	Top3	MRR	Top1	Top2	Top3	MRR
Our Approach	Hadoop	12.20	25.61	36.59	22.56	29.27	54.88	65.85	45.73	45.12	69.51	76.83	59.76
	Hive	8.70	20.65	28.26	17.21	17.39	36.96	50.00	31.52	30.43	55.43	72.83	48.73
	Pig	9.09	22.73	27.27	17.42	45.45	68.18	72.73	58.33	77.27	90.91	90.91	84.09
	HBase	19.19	25.25	32.32	24.58	46.46	58.59	70.71	56.67	64.65	78.79	86.87	74.41
	Derby	8.82	11.76	23.53	14.22	26.47	38.24	58.82	39.22	47.06	70.59	79.41	61.76
	Zookeeper	6.67	20.00	26.67	15.56	13.33	33.33	46.67	27.78	33.33	53.33	66.67	47.78
Random Selection	Hadoop	4.93	9.40	13.62	8.57	14.40	26.63	36.34	23.75	25.18	42.91	54.90	38.04
	Hive	6.48	12.71	18.47	11.51	15.39	28.86	39.67	25.73	27.62	46.60	59.63	41.45
	Pig	5.77	11.36	16.73	10.36	16.59	29.95	41.45	27.11	34.91	57.00	71.95	50.94
	HBase	6.88	13.00	18.72	11.85	18.75	32.88	44.22	29.59	31.34	51.24	64.58	45.74
	Derby	7.71	15.03	21.71	13.59	23.35	40.03	52.29	35.78	43.44	64.65	75.26	57.58
	Zookeeper	5.07	11.33	18.13	10.47	19.07	32.53	46.53	30.47	33.13	53.00	68.07	48.09

Carlo simulation with 100 iterations for each case. Table 5.3 presents Top-k accuracy and MRR of both our algorithm and Monte Carlo simulation for six-month absence limit, while Table 5.4 presents the the same for one-year absence limit.

In these tables, when we look at the small-scale projects, which are Pig, Derby and Zookeeper, the accuracy of our approach and the accuracy of the Monte Carlo simulation is close for many cases. Even in some cases, the randomly selected developers are more accurate than our approach. Therefore, suggesting replacement in small-scale projects is not meaningful because the number of developers is so small, and even random selection can work.

Algorithm 9 Finding if Balanced or Hero Team (or Project) according to the Pareto principle [31]

```
1: function BalancedOrHeroPareto(devToCommitCount)
2:     numCoveredCommits ← 0
3:     numCoveredDevs ← 0
4:     numAllDevs ← devToCommitCount.keys().length()
5:     commitCounts ← SortDescending(devToCommitCount.values())
6:     numAllCommits ← Sum(commitCounts)
7:
8:     for commitCount in commitCounts do
9:         numCoveredCommits ← numCoveredCommits + commitCount
10:        numCoveredDevs ← numCoveredDevs + 1
11:        if numCoveredDevs / numAllDevs >= 0.2 then
12:            break
13:
14:    if numCoveredCommits / numAllCommits >= 0.8 then
15:        return "hero"
16:    else
17:        return "balanced"
```

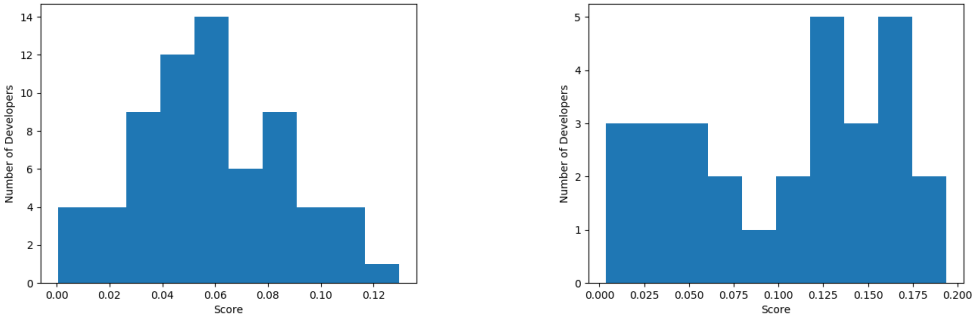
5.2.3 Results for Knowledge Distribution (RQ 3)

To compare our results, we implemented the algorithm of Agrawal et al.[31] as given in Algorithm 9 by using the explanations in their paper. First, it finds the number of commits made by the 20% or more of the top committers. Then, it returns *hero* if the covered commits are 80% or more of all commits, otherwise returns *balanced*.

For each project, we generated results of our algorithm (Shapiro-Wilk Test on file coverage distribution given as in Algorithm 8) and the algorithm of Agrawal et al. (Pareto principle on commit counts given in Algorithm 9). While deciding if the project is *hero* or *balanced* project with both algorithms, we used the sliding window approach as explained in Section 5.1. Thus, we used the commits made in the sliding window period (e.g., the last 365 days when the sliding window size is 365 days) for each iteration. Table 5.5 shows the results for our approach and the Pareto principle approach. For example, our approach labeled the knowledge distribution of the Hive project as *balanced* for 73.12% of all iterations when

Table 5.5: Results for balanced and hero projects (in percent)

Sliding Window Size	Project	Shapiro-Wilk Test		Pareto Principle		Accuracy
		Balanced	Hero	Balanced	Hero	
six months	**Hadoop**	64.30	35.70	100.00	0.00	64.30
	Hive	73.12	26.88	90.31	9.69	61.12
	Pig	91.38	8.62	90.27	9.73	85.91
	HBase	74.15	25.85	89.94	10.06	65.70
	Derby	93.14	6.86	98.43	1.57	91.95
	Zookeeper	76.63	23.37	96.47	3.53	75.16
one year	**Hadoop**	74.51	25.49	100.00	0.00	74.51
	Hive	69.76	30.24	89.78	10.22	61.94
	Pig	95.03	4.97	87.09	12.91	85.22
	HBase	73.36	26.64	64.47	35.53	66.12
	Derby	93.80	6.20	99.25	0.75	94.08
	Zookeeper	85.80	14.20	99.73	0.27	85.52

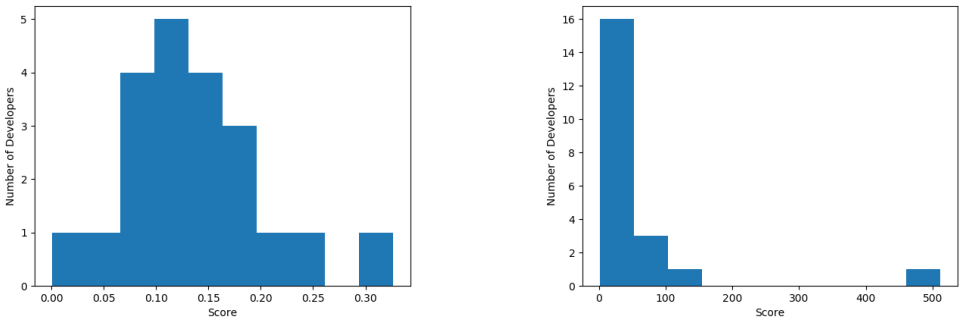


(a) Balanced (Hadoop, 06 March 2017) (b) Hero (Hadoop, 30 August 2012)

Figure 5.4: File coverage histogram examples for balanced and hero projects according to our approach (Shapiro-Wilk) (Sliding window size is one year)

the sliding window size is six months. Accuracy is the ratio of the number of iterations in which both algorithms produced the same label over the number of all the iterations.

Figure 5.4a presents the histogram of the file coverage distribution of Hadoop developers in 06 March 2017, and Figure 5.4b presents the same histogram for 09 August 2012. The first one labeled as *balanced* and the latter one labeled as *hero* by our algorithm. It is clearly seen that the balanced histogram is closer to a normal distribution than the other one.



(a) File Coverage (b) Number of Commits

Figure 5.5: An example day (Hive, 14 January 2014) labeled as balanced by the Shapiro-Wilk test and hero by the Pareto principle. (Sliding window size is one year)

Figure 5.5 shows histograms for file coverage distribution and commit count distribution of Hive developers on 14 January 2014. Our algorithm labels that day as *balanced*, but the Pareto principle algorithm labels it as *hero* project. Since the developers who made a small number of commits are dominant and one developer made almost 500 commits, the Pareto principle algorithm labels that as *hero*. However, the file coverage distribution seems like a normal distribution since our algorithm is not only affected by the number of commits but also affected by the co-changed files of different change sets (developers can reach the files that they did not directly modify) and the recency of the contribution (recent commits are more important). Therefore, according to our algorithm, the team does not heavily depend on the top committer and that the other developers have enough knowledge about the files changed in the last year.

Chapter 6

Manager Dashboard Tool (Proof of Concept)

We implemented a proof of concept tool in Python using Plotly Dash framework[1]. The tool provides a useful user interface to traverse the results of our experiments over time. Traversing day by day (forward and backward) and selecting a specific date in the whole experiment is possible. For the selected day, it provides the following information:

- A summary which shows the list of developers sorted by name and their scores for each key developer category.
- Number of all files in the project, number of reachable files, number of rarely reached files, number of developers and knowledge distribution label ("balanced" or "hero").
- A Venn diagram showing the key developers for different categories.
- For each key developer category (jacks, mavens and connectors), list of developers sorted by their scores.

[1] https://plotly.com/dash/

- For each key developer category, a plot showing all developers in that category with their scores over time (time period can be changed in the user interface).

Fig. 6.1 is a screenshot of experiment selection, date selection, summary and Venn diagram parts of the tool. Fig. 6.2 shares a screenshot of the connector division which includes the list of connectors for the selected date and the past betweenness centrality scores of these connectors. It also shows the replacement division which includes the list of the leaving developers for the selected date and three replacements for each leaving developer with corresponding overlapping knowledge ratios. The source code is available online (See Appendix B). With some configurations and additions, this proof of concept tool can be used for monitoring development projects day by day by the practitioners.

Figure 6.1: Screenshot of selection parts, summary and Venn diagram (developer names are painted black)

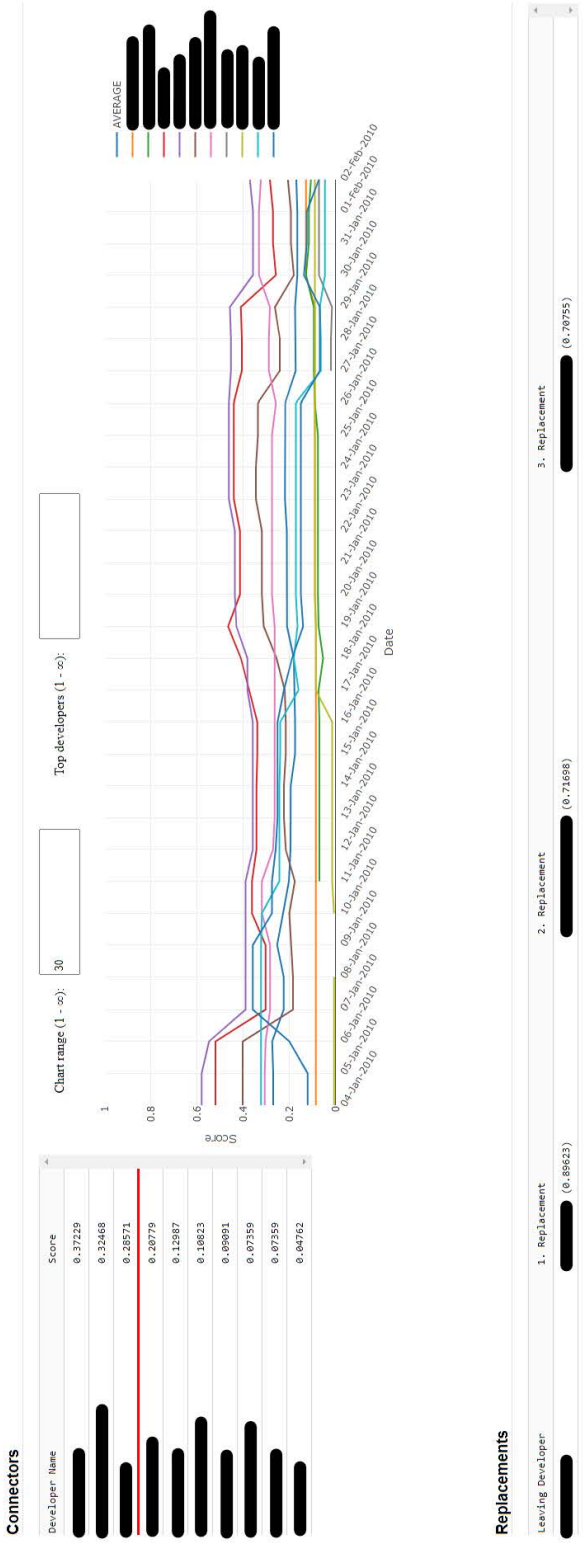


Figure 6.2: Screenshot of connectors and replacements divisions (developer names are painted black)

Chapter 7

Discussion

In this section, we discuss the RQs defined in Section 1, the implications of our study and the scalability of our algorithms.

7.1 Research Questions

In this study, we investigated three RQs about how to identify key developers, how to find replacements for leaving developers and how to decide whether knowledge distribution in a team is balanced or not. We discuss the implications of them one by one.

7.1.1 How to identify key developers (RQ1)

To identify key developers, we defined three different kinds: jack, maven and connector. Then, we shared algorithms to identify them using artifact traceability graphs. In the literature, there are studies on core and periphery [27, 21], core, active, occasional and rare [28], core, external and mutant [29], key [30], hero [31] and elite [32] developers in OSS projects and LTC type of developers [33]. These

developer definitions are closely related to our general key developer definition, even though they are not exactly the same.

The main difference in our study is the subcategorization of key developers. The others define either just one key developer category or categories for all developers and one of the categories is key developer. In our work, we shared different key developer definitions using different perspectives and separate algorithms for each of them. Jack, maven and connector are novel definitions for key developers.

Another difference is the evaluation setup in our study. We use a sliding window approach and find key developers for each day, not just for one day in the project history. Also, our validation approach is different. Since there is no real-life label for key developers, researchers followed different approaches. Agrawal et al. [31] used the Pareto principle in their definition from previous works [34, 35]. These previous studies just investigated the Pareto principle on commit counts of the developers without providing a validation showing that the Pareto principle is suitable for deciding whether a project is a hero project or not, and Agrawal et al. [31] used the same definition. Bella, Sillitti and Succi [28] used clustering, and thus their validation is only about performing a good enough clustering. Oliva et al. [30] performed a case study on a small project with 16 developers. Crowston et al. [27] somehow used the labels on SourceForge[1]. Joblin et al. [21] made a survey with 166 developers, and their results are not available now. Padhye, Mani and Sinha [29] and Wang et al. [32] defined core/elite developers using their access rights to repositories. Zhou and Mockus [33] focused on how a new joiner becomes an LTC, not validating the importance of LTC developers. In our study, we perform case studies using six OSS projects which are different in scale. Then, we validate our approach using a different data source that is not used in our models, the comments to the issues in Jira. As the baseline approach, we performed Monte Carlo simulations to detect key developers randomly. As a result, our approach is more successful than the random model for all cases (See Section 5.2.1).

[1]https://sourceforge.net/

7.1.2 How to find replacements for leaving developers (RQ2)

We proposed to use overlap ratios of reachable files (i.e., overlapping knowledge) to replace a leaving developer. Rigby et al. [11] proposed a successor (replacement) recommendation approach while investigating turnover-induced knowledge loss in software projects. They used random selection as their baseline approach and compared their results. Since they did not share the implementation of their approach, we were not able to use their approach as a baseline. Also, we used data from SQL tables [1], not from git repositories and their approach depends on *git blame* command. So, performing experiments with both their model and our model is not possible if we do not change our data source. However, we performed a Monte Carlo simulation (See Section 5.2.2) as a baseline approach. Also, our validation for replacement recommendation is similar to theirs [11], and we shared the relevant details more explicitly. Results show that our approach works better than the random case for large-scale projects. For the small-scale projects, the random case is as accurate as our approach for some cases (See Section 5.2.2).

7.1.3 How to decide whether knowledge distribution in a team is balanced or not (RQ3)

We assumed that the knowledge distribution of a balanced team would follow a normal distribution. We proposed that the file coverage ratio represents the developer's knowledge about the project. Then, we used Shapiro-Wilk [42] normality test on file coverage ratios of all active developers in the project to decide whether knowledge distribution in a team is balanced or not. Agrawal et al.[31] classified a project as a hero project if 20% of the developers made 80% of the contributions, otherwise they classified the project as a non-hero project. To compare our results, we implemented their approach. Using both algorithms, we labeled projects hero or balanced (non-hero) for each day of their history. Labels

of both algorithms overlapped 61-94% in all histories of the six OSS projects that we used in our case studies (See Section 5.2.3).

7.2 Scalability

We presented eight different algorithms while addressing the RQs. We discuss their time complexity one by one (N is the number of nodes, E is the number of edges, D is the number of developers, F is the number of files, C is the number of change sets and I is the number of issues):

- Algorithm 1 (Finding Reachable Files): First it finds the list of developers by looking at every node in the graph, and it takes $\Theta(N)$ time. Then it runs a DFS for each developer with a distance limit of 10. Since a single DFS may search all of the graph and take *O(N+E)* time, the time complexity of the algorithm is *O(D(N+E))*.

- Algorithm 2 (Finding Jacks): First, it calls Algorithm 1 to find reachable files of developers, which takes *O(D(N+E))* time. Then, it loops over the developer and reachable files pairs and calculates file coverage. So, this loop takes $\Theta(D)$ time, and the overall time complexity of this algorithm is *O(D(N+E)+D) = O(D(N+E+1)) = O(D(N+E))*.

- Algorithm 3 (Finding Rarely Reached Files): First, it calls Algorithm 1 to find reachable files of developers, which takes *O(D(N+E))* time. Then, it inverts the mapping by looping through all developers and their reachable files. Since, theoretically, any file can be reached by every developer, this step takes *O(DF)*. Afterward, it loops over files and checks if the file is reached by only one developer or not, which takes *O(F)* time. *O(DF)* and *O(F)* increase run time but do not affect asymptotic analysis. Because N is an upper boundary for F, *O(DF)* is *O(DN)* and *O(F)* is *O(N)*. So, the overall time complexity of this algorithm is *O(D(N+E)+DN+N) = O(2DN+DE+N) = O(N(2D+1)+DE) = O(ND+DE) = O(D(N+E))*.

- Algorithm 4 (Finding Mavens): First, it calls Algorithm 3 to find rarely reached files of developers, which takes *O(D(N+E))* time. Then, it finds the number of all rarely reached files, which is a simple summation of the number of rarely reached files of developers and takes $\Theta(D)$ time. Afterward, it loops over the developer and rarely reached files pairs to calculate the mavenness score, which takes $\Theta(D)$ time. So, the overall time complexity of this algorithm is *O(D(N+E)+D+D) = O(D(N+E+2) = O(D(N+E))*.

- Algorithm 5 (Calculating RSRD): The first outer for loop runs a DFS for each developer with a depth limit of 4. A single DFS may search all of the graph and take *O(N+E)*. If DFS dominates, it takes *O(D(N+E))*. Another possibility is that the inner for loop can dominate. In that case, we can look at the number of total possible simple paths between two developers. Because of the graph structure defined in Section 3.1, developers are only connected to change sets and change sets can be connected to issues and files. A path with a length of 4 between two developers has to go through two change sets and a file node or issue node. Thus, there might be $O(DC(F+I)(C-1)(D-1)) = O(D^2C^2(F+I))$ different paths. So, the first nested two for loops take $O(max(D(N+E), D^2C^2(F+I)))$ time. The second two nested for loops also do $O(D^2C^2(F+I))$ summations because they iterate over all possible paths. So, they take $O(D^2C^2(F+I))$ time. The overall complexity of the algorithm $O(max(D(N+E), D^2C^2(F+I)))$. Since C, F and I cannot be more than N, a looser boundary is also $O(max(D(N+E), D^2N^3))$ (Because D is very small in general, we do not replace it with N).

- Algorithm 6 (Finding Connectors): First, it calls Algorithm 5 to find RSRD values, which takes $O(max(D(N+E), D^2C^2(F+I)))$ time. Then, it creates a developer graph with nodes for developer nodes and edges for RSRD values between developers. The number of nodes will be $\Theta(D)$ and the number of edges will be $O(D^2)$. So, this step takes $O(D^2)$ times. Afterward, it computes the betweenness centralities of all nodes in the developer graph. Betweenness centrality calculations are limited by the number of nodes plus the number of edges [40], which corresponds to $O(D + D^2) = O(D^2)$

in the developer graph. So, the overall complexity of this algorithm is $O(max(D(N+E), D^2C^2(F+I))+D^2) = O(max(D(N+E), D^2C^2(F+I)))$.

- Algorithm 7 (Finding Replacements): First, it calls Algorithm 1 to find reachable files for each developer, which takes *O(D(N+E))* time. Then, it finds other developers, which takes $\Theta(D)$ time. Afterward, it loops over the other developers to find overlapping knowledge of the leaving developer and others. Time complexity of intersection operation of s and t lists takes *O(max(len(s), len(t)))*.[2] s and t are the list of reachable files in our case. They cannot be more than F, and F cannot be more than N. So, the overall complexity of this algorithm is $O(D(N + E) + D + N) = O(DN + DE + D + N) = O(N(D + 1) + D(E + 1)) = O(ND + DE) = O(D(N + E))$.

- Algorithm 8 (Finding Knowledge Distribution): First, it calls Algorithm 2 to find jacks, which takes *O(D(N+E))* time. Then, it performs the Shapiro-Wilk test over file coverage scores of developers. We use *scipy* package[3] in our implementation (See Appendix B). It sorts the inputs ($\Theta(D \log D)$) and calls the algorithm of Royston [46] ($\Theta(D)$). Because N is an upper boundary for D, $\Theta(D \log D)$ is less than $\Theta(DN)$. So, the overall complexity of this algorithm is $O(D(N + E) + D \log D) = O(D(N + E))$.

All algorithms depend on DFS because they use Algorithm 1 or Algorithm 5. In Algorithm 1, distance limit (threshold) reduces run time but it is not explicitly given in our complexity analysis. DFS depth changes for each search because the distance of the traversed edges depends on their recency. Distance of an edge can be at least 1 when the recency is maximum, which is 1 (See Equation 3.1 and Equation 3.2). Therefore, the depth of DFS cannot be more than 12 (in the worst case, the first and the last edges can have 0 distance if they connect developers and change sets) in Algorithm 1. Also, the depth can be a minimum of 2 when recency values of all edges of the change sets of the developer are less than 0.1, which correspond to distance values more than 10. In that case, it traverses one edge with 0 distance for going to a change set node from a developer node and

[2]https://wiki.python.org/moin/TimeComplexity
[3]https://www.scipy.org/

one edge with 10+ distance for exceeding the threshold. So, actually, it does not traverse the whole graph because depths change between 2 and 12. Similarly, in Algorithm 5, we used a constant depth limit of 4 while finding paths between two developers. Again, it does not traverse the whole graph. Since we have such limits, the DFS parts of our algorithms are faster than we discussed above. However, the analysis above is still accurate because we defined upper limits with Big-O notation.

Table 7.1 presents the average number of artifacts (N, E and D) mentioned above and the average time taken to find jacks, mavens, connectors and knowledge distribution in the experiments. For example, the Hive project has 3003 iterations (its window slides 3002 times), and the average number of nodes in the whole artifact graph is 3237 per iteration. Each iteration corresponds to a day in their history. Table 7.2 presents the average number of artifacts (N, E and D) mentioned above and the average time taken to find replacements. Since we measured the performance of this category only when a developer left the project, the numbers of iterations are smaller. Even for a large-scale project like Hadoop, calculating each category takes less than 1 second a day on average with our Python implementation (See Appendix B) working on an ordinary laptop.

Table 7.1: Average number of artifacts and average time taken in jack, maven, connector and knowledge distribution experiments.

Project	**Average number of artifacts**			**Average time taken (seconds)**				**Number of iterations**
	Developer nodes (D)	**All nodes (N)**	**Edges (E)**	**Jacks**	**Mavens**	**Connectors**	**Balanced or Hero**	
Hadoop	50	5956	11822	0.6803	0.7325	0.9974	0.6862	2742
Hive	35	3237	6138	0.4574	0.5163	0.6323	0.4640	3003
Pig	9	1046	1705	0.0519	0.0618	0.0727	0.529	2774
HBase	32	2984	6221	0.4222	0.4651	0.6050	0.4223	3521
Derby	11	1542	2874	0.0901	0.1086	0.1456	0.0931	4465
Zookeeper	8	374	607	0.0185	0.0213	0.0310	0.0193	3302

Table 7.2: Average number of artifacts and average time taken in the replacement experiments.

Project	Average number of artifacts			Average time taken (seconds)	Number of iterations
	Developer nodes (D)	Nodes (N)	Edges (E)		
Hadoop	48	5922	11858	0.6227	89
Hive	63	4611	8857	0.6747	106
Pig	11	1205	2012	0.0555	25
HBase	49	3718	7595	0.6080	105
Derby	13	2130	4227	0.1509	37
Zookeeper	7	318	485	0.0115	17

7.3 Practical Implications

We proposed algorithms to identify key developers in three types, an algorithm to find replacements for a leaving developer and an approach to evaluate if knowledge distribution in a project is balanced or not. Knowing the indispensable developers in the project can help the manager for future decisions. For example, if a developer is the only connector of the team, the manager would not want her/him to leave the project because (s)he may be the only one who has connections with other teams, and collaboration with other teams would be at risk if (s)he leaves. Also, knowing the possible replacements can make the manager's job easier to decide who will take over the jobs when one leaves. Instead of taking over the jobs, recommended replacements can mentor other developers who take the jobs. Moreover, the manager can take precautions to distribute knowledge amongst developers when it is not balanced.

All three scenarios that we worked on in this study are to help the team and the managers to reduce the risk for the software development process. However, our approaches have an explicit shortcoming. They are not suitable for relatively inactive projects with a small number of developers. The accuracy and MRR results for replacement recommendation in Section 5.2.2 explicitly illustrate the cases that the random selection works as accurately as our replacement algorithm. Especially for Derby and Zookeeper cases, the random selection results and the results of our approach are very close to each other. In a few cases, the random selection is even better. For identifying key developers and evaluating

the knowledge distribution in the project, this shortcoming is not that explicitly shows itself in our results, but it is kind of obvious that identifying key developers, recommending replacements or evaluating knowledge distribution in a small team does not mean much since everybody knows everybody in a small team, even random selection would do the work.

The algorithms that we proposed are implemented in Python for our experiments. The implementation is shared online in a GitHub repository (See Appendix B). Thus, future researchers can replicate the results of our experiments. Also, this implementation proves that our algorithms are scalable as we mentioned above. Even though Hadoop, Hive and HBase are very active and large-scale projects, any algorithm proposed in this study takes less than 1 second a day even using an ordinary laptop. Therefore, the algorithms are practically usable at the present by the managers of these projects.

We provided a prototype tool to traverse through the results of the experiments. It is a proof of concept tool for a manager dashboard in the software development projects. So, with a little configuration on the tool, the metrics produced by our algorithms can be used by managers of projects to monitor the state of developers. Especially for a team of globally distributed software engineers, having a such system would ease the management.

Lastly, comparing developers to each other may cause unexpected results. For example, developers would be inclined to change the files that have not been changed for a long time if they know the maven definition. Many companies and organizations have some metrics to evaluate the benefits that developers provide to the project. An OSS example that uses such metrics is Eclipse Sirius project[4]. Also, OpenHub provides a webpage[5] for the statistics of the same project. They generally share explicit outcomes such as commit count and frequency. Therefore, developers already know that they are evaluated somehow. The difference in our study is that our metrics are not that obvious to calculate. They implicitly

[4]https://projects.eclipse.org/projects/modeling.sirius/who (Accessed on 24 Sep 2020)

[5]https://www.openhub.net/p/eclipse_sirius/contributors/summary (Accessed on 24 Sep 2020)

evaluate developers from different perspectives. For example, co-changed of files and change recency affects our metrics.

Chapter 8

Threats to Validity

Construct validity is about how the operational measures in the study represent what is investigated according to the RQs [47]. We used datasets from another study [1], and their mining process can potentially affect our results. To reduce the threat caused by the data mining process, we eliminated the possible problems (e.g., we corrected author names manually by looking at their names and email addresses.) in preprocessing (See Section 4.2). However, there might still be problems related to data integrity.

Internal validity concerns if the causal relations are examined or not [47]. While building the graphs and defining algorithms, we made many decisions related to thresholds including, but not limited to:

- Choosing 50 as the limit for the number of files added or modified in a change set
- Choosing 10 as the distance threshold in file reachability
- Using six-month and one-year sliding windows in the experiments
- Using 180 days (six months) and 365 days (one year) as absence limit while detecting leaving developers

We tried various options and made the final decisions after evaluating their results. In the corresponding sections of this study, we shared the justifications behind these decisions. For example, we chose 10 as the distance threshold since it corresponds to 90% of the covered time in the graph due to the nature of the distance formula and it is a great trade-off point.

We measured project knowledge depending on source codes and assumed that the knowledge of a developer decreases in time (recency). But there is a significant point against that: documentation. Software projects generally have documentation, and it preserves the knowledge at some level if it is maintained well. Even if the documentation is not sufficient in a project, still it means that some parts of the know-how of the project are preserved. However, having documentation would not decrease the values of the key developers in a project. After a while, developers forget some parts of the source code if they do not work on it. Thus, we believe, using code changes with their recency is still a good way to measure knowledge.

The potential errors in the implementation of our approach threaten the validity of our results. We benefited from a stable graph package NetworkX [39] in our operations and used its methods whenever possible, for example, betweenness centrality calculations and DFS for finding paths between developers. To prevent potential bugs, we performed multiple code review sessions with three researchers besides the authors of the study. Also, we shared the implementation online (See Appendix B) for replicability of the results.

We used developer comments in issue tracking systems to validate our approach, however, we do not claim that the number of comments shows the key developers in a project. We just claim that there should be a correlation between the top commenters and the key developers in the same time period. Then, we used this idea to show that our approach produced more logical results than the random case with a Monte Carlo simulation.

Also, we evaluated our replacement validation algorithm using a very similar way to the way of Rigby et al. [11]. It is a heuristic evaluation, thus our accuracy

could be different with real-life labels for the successors (replacements) of the leaving developers. And again, we showed that our algorithms perform better than the random case, especially for large-scale projects. Moreover, we used the algorithm of Agrawal et al.[31] to show that our knowledge distribution algorithm is mostly consistent with their approach. So, the real-life labels for balanced and hero teams could give different results. Using existing approaches in the literature helped us to reduce these threats to our validity and compare our results.

External validity concerns about generalization of the findings in studies [47]. In our case studies, we used six different OSS projects. Even though we did not conduct a case study in an industrial company, we selected projects from Apache[1], a 20-year established foundation. Also, the sizes of the projects are different as seen in Table 4.1. Although we believe that we have enough data for an initial assessment, in the future, we need to run our algorithms in more OSS and industrial datasets.

[1]https://www.apache.org/

Chapter 9

Conclusion and Future Work

In this study, we constructed artifact traceability graphs by using software artifacts and their relations. Then, we proposed different categories for key developers in software development projects: *jacks*, *mavens* and *connectors*. To identify the developers in these subcategories of key developers, we proposed separate algorithms using artifact traceability graphs. Also, we proposed an algorithm to find replacements (successors) for leaving developers, and another algorithm to evaluate the knowledge distribution amongst developers in a team.

To evaluate our proposed algorithms, we conducted case studies on six OSS projects (Hadoop, Hive, Pig, HBase, Derby and Zookeeper). Since there was no labeled data for the key developer categories, we used developers' comments in issue tracking systems to validate our results. The key developers found by our model were compatible with the top commenters up to 98%. Also, we validated our replacement recommendation algorithm similar to the approach of Rigby et al.[11]. For different projects and parameters, its accuracy changes up to 91%. Moreover, we validated our knowledge distribution evaluation algorithm by comparing the approach of Agrawal et al. [31]. Our recommended labels were compatible up to 94% with the labels produced by the algorithm of Agrawal et al. [31] .

The results indicated that our approaches have promising results to identify key developers in software projects, to recommend replacements for leaving developers and to evaluate knowledge distribution amongst developers. We can summarize the contributions of our study as follows:

- We offered a novel categorization for the key developers inspired by the kinds of humans who turn ideas into epidemics in The Tipping Point by Gladwell [7].
- For each of the three key developer categories (jacks, mavens and connectors), we proposed an algorithm using traceability graphs (network) of software artifacts.
- The findings of this study might shed light on the truck-factor problem. Key developers might help to find the truck factor of the projects.
- Identifying key developers in a software project might help the software practitioners in making managerial decisions.
- We provided a proof of concept tool to traverse through project histories and observe key developers with their corresponding scores.
- Evaluating knowledge distribution amongst developers might help managers to take early precautions.
- Finding a replacement in a large team might help for managerial decisions in case of unexpected and sudden leaves of developers.

The followings are the possible future directions of this study:

- **Enriching the Artifact Traceability Graph:** Our algorithms heavily depend on the structure and the content of the artifact traceability graph. There are different ways to change or improve the graph:
  - We used change sets, source files and issues in the traceability graph. The graph can be enriched by adding extra nodes for other artifacts

such as design documents and hardware related documents depending on the project.

  - The edges in the graph can be enriched by adding new relations among the artifacts. For instance, the dependencies between source files can be represented as edges among file nodes. The recency of such edges can be the highest value since they are directly connected in the source code.
  - In multidisciplinary projects, there might be artifacts from different domains. In that case, the artifact traceability graphs represent the structures from these different disciplines (i.e., domains). When having a problem with a part of the project, identifying responsible discipline or the group of people might be possible using the traceability graph.

  Another future direction involving all the items above would be inspecting how such additions affect our algorithms and the results.

- **Experiments on Industrial Datasets:** In our study, we performed experiments on OSS datasets. An evaluation of our algorithms on industrial datasets is required. Besides running our algorithms on industrial datasets, validating the results by interviewing project stakeholders and creating a labeled dataset for the introduced key developer types might be possible directions.

- **Tool Extension:** We developed a proof of concept tool to show how managers would see the results of our RQs throughout the project history. The current capability of the tool is visualizing the generated results. The tool can be potentially extended as a plugin tool to GitHub. In that case, the proposed algorithms will be available to GitHub users to evaluate their projects from our perspectives.

- **Developing Predictive Models:** Our work is an observational study. We observe the current situation of the project by looking at its history. Developing predictive models for the RQs in this is a possible direction. For example, potential future key developers can be predicted by mining project history.

Bibliography

[1] M. Rath and P. Mäder, "The seoss 33 dataset—requirements, bug reports, code history, and trace links for entire projects," *Data in brief*, vol. 25, p. 104005, 2019.

[2] M. E. Conway, "How do committees invent," *Datamation*, vol. 14, no. 4, pp. 28–31, 1968.

[3] J. D. Herbsleb and R. E. Grinter, "Splitting the organization and integrating the code: Conway's law revisited," in *Proceedings of the 21st international conference on Software engineering*, pp. 85–95, 1999.

[4] A. Mockus, "Organizational volatility and its effects on software defects," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 117–126, 2010.

[5] H. A. Cetin, "Identifying the most valuable developers using artifact traceability graphs," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1196–1198, 2019.

[6] H. A. Çetin and E. Tüzün, "Identifying key developers using artifact traceability graphs," in *Proceedings of the 16th ACM International Conference on Predictive Models and Data Analytics in Software Engineering*, pp. 51–60, 2020.

[7] M. Gladwell, *The tipping point: How little things can make a big difference*. Little, Brown, 2006.

[8] G. Avelino, L. Passos, A. Hora, and M. T. Valente, "A novel approach for estimating truck factors," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pp. 1–10, IEEE, 2016.

[9] T. Fritz, G. C. Murphy, E. Murphy-Hill, J. Ou, and E. Hill, "Degree-of-knowledge: Modeling a developer's knowledge of code," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 2, pp. 1–42, 2014.

[10] V. Cosentino, J. L. C. Izquierdo, and J. Cabot, "Assessing the bus factor of git repositories," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 499–503, IEEE, 2015.

[11] P. C. Rigby, Y. C. Zhu, S. M. Donadelli, and A. Mockus, "Quantifying and mitigating turnover-induced knowledge loss: case studies of chrome and a project at avaya," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 1006–1016, IEEE, 2016.

[12] G. Avelino, E. Constantinou, M. T. Valente, and A. Serebrenik, "On the abandonment and survival of open source projects: An empirical investigation," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1–12, IEEE, 2019.

[13] M. Ferreira, T. Mombach, M. T. Valente, and K. Ferreira, "Algorithms for estimating truck factors: a comparative study," *Software Quality Journal*, vol. 27, no. 4, pp. 1583–1617, 2019.

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

[15] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 931–940, IEEE, 2013.

[16] G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "Who is going to mentor newcomers in open source projects?," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pp. 1–11, 2012.

[17] M. Nassif and M. P. Robillard, "Revisiting turnover-induced knowledge loss in software projects," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 261–272, IEEE, 2017.

[18] A. Amirfallah, F. Trautsch, J. Grabowski, and S. Herbold, "A systematic mapping study of developer social network research," *arXiv preprint arXiv:1902.07499*, 2019.

[19] J. Wu and K. Y. Goh, "Evaluating longitudinal success of open source software projects: A social network perspective," in *2009 42nd Hawaii International Conference on System Sciences*, pp. 1–10, IEEE, 2009.

[20] T. Kakimoto, Y. Kamei, M. Ohira, and K. Matsumoto, "Social network analysis on communications for knowledge collaboration in oss communities," in *Proceedings of the International Workshop on Supporting Knowledge Collaboration in Software Development (KCSD'06)*, pp. 35–41, Citeseer, 2006.

[21] M. Joblin, S. Apel, C. Hunsen, and W. Mauerer, "Classifying developers into core and peripheral: An empirical study on count and network metrics," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 164–174, IEEE, 2017.

[22] M. Y. Allaho and W.-C. Lee, "Analyzing the social ties and structure of contributors in open source software community," in *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pp. 56–60, 2013.

[23] M. V. Kosti, R. Feldt, and L. Angelis, "Archetypal personalities of software engineers and their work preferences: a new perspective for empirical studies," *Empirical Software Engineering*, vol. 21, no. 4, pp. 1509–1532, 2016.

[24] J. Cheng and J. L. Guo, "Activity-based analysis of open source software contributors: roles and dynamics," in *2019 IEEE/ACM 12th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pp. 11–18, IEEE, 2019.

[25] R. Milewicz, G. Pinto, and P. Rodeghero, "Characterizing the roles of contributors in open-source scientific software projects," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 421–432, IEEE, 2019.

[26] M. Ortu, T. Hall, M. Marchesi, R. Tonelli, D. Bowes, and G. Destefanis, "Mining communication patterns in software development: A github analysis," in *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, pp. 70–79, 2018.

[27] K. Crowston, K. Wei, Q. Li, and J. Howison, "Core and periphery in free/libre and open source software team communications," in *Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS'06)*, vol. 6, pp. 118a–118a, IEEE, 2006.

[28] E. Di Bella, A. Sillitti, and G. Succi, "A multivariate classification of open source developers," *Information Sciences*, vol. 221, pp. 72–83, 2013.

[29] R. Padhye, S. Mani, and V. S. Sinha, "A study of external community contribution to open-source projects on github," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 332–335, 2014.

[30] G. A. Oliva, J. T. da Silva, M. A. Gerosa, F. W. S. Santana, C. M. L. Werner, C. R. B. de Souza, and K. C. M. de Oliveira, "Evolving the system's core: a case study on the identification and characterization of key developers in apache ant," *Computing and Informatics*, vol. 34, no. 3, pp. 678–724, 2015.

[31] A. Agrawal, A. Rahman, R. Krishna, A. Sobran, and T. Menzies, "We don't need another hero?: the impact of heroes on software development," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pp. 245–253, ACM, 2018.

[32] Z. Wang, Y. Feng, Y. Wang, J. A. Jones, and D. Redmiles, "Unveiling elite developers' activities in open source projects," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 3, pp. 1–35, 2020.

[33] M. Zhou and A. Mockus, "What make long term contributors: Willingness and opportunity in oss community," in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 518–528, IEEE, 2012.

[34] M. Goeminne and T. Mens, "Evidence for the pareto principle in open source software activity," in *the Joint Porceedings of the 1st International workshop on Model Driven Software Maintenance and 5th International Workshop on Software Quality and Maintainability*, pp. 74–82, Citeseer, 2011.

[35] K. Yamashita, S. McIntosh, Y. Kamei, A. E. Hassan, and N. Ubayashi, "Revisiting the applicability of the pareto principle to core development teams in open source software projects," in *Proceedings of the 14th International Workshop on Principles of Software Evolution*, pp. 46–55, 2015.

[36] K. Yamashita, Y. Kamei, S. McIntosh, A. E. Hassan, and N. Ubayashi, "Magnet or sticky? measuring project characteristics from the perspective of developer attraction and retention," *Journal of Information Processing*, vol. 24, no. 2, pp. 339–348, 2016.

[37] L. C. Freeman, "Centrality in social networks conceptual clarification," *Social networks*, vol. 1, no. 3, pp. 215–239, 1978.

[38] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *Proceedings of the 2006 international workshop on Mining software repositories*, pp. 137–143, 2006.

[39] "Networkx." https://networkx.org/. (Accessed on 28 Dec 2020).

[40] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of mathematical sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[41] E. Sülün, E. Tüzün, and U. Doğrusöz, "Reviewer recommendation using software artifact traceability graphs," in *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*, pp. 66–75, 2019.

[42] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3/4, pp. 591–611, 1965.

[43] N. M. Razali, Y. B. Wah, *et al.*, "Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests," *Journal of statistical modeling and analytics*, vol. 2, no. 1, pp. 21–33, 2011.

[44] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: a case study at google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pp. 181–190, 2018.

[45] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 202–212, 2013.

[46] P. Royston, "Remark as r94: A remark on algorithm as 181: The w-test for normality," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 44, no. 4, pp. 547–551, 1995.

[47] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical software engineering*, vol. 14, no. 2, p. 131, 2009.

Appendix A

Data

In the experiments, we used the data shared by Rath and Mader [1]. The data is available on Harvard Dataverse.[1,2]

[1] https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/PDDZ4Q (Accessed on 24 Sep 2020)

[2] https://bit.ly/2wukCHc (The link shared by the original study [1]) (Accessed on 24 Sep 2020)

Appendix B

Source Code

The source code used in this study is shared on GitHub.[1]

[1] https://github.com/hacetin/msc-thesis (Accessed on 27 Dec 2020)