The Human and Social Aspects of Software Engineering

By

MOHAMMAD GHAREHYAZIE

B.S. (Sharif University of Technology) 2007
M.S. (Sharif University of Technology) 2010
M.S. (University of California, Davis) 2012

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Vladimir Filkov, Chair

Raissa D'Souza

Premkumar Devanbu

Committee in Charge

2016

Contents

Mohammad Gharehyazie
June 2016
Computer Science

The Human and Social Aspects of Software Engineering

Abstract

Researchers in the field of Empirical Software Engineering have been studying software development for a long time. While many of these studies have focused on the artifact of this process, only some have discussed issues that revolve around developers. Even among those studies, most of them view the software development either as a whole (black box) or as a number of individuals. However, we all know that good software is a result of ”teamwork” and ”collaboration”, thus how developers - and other individuals involved in the process - interact with each other and its dynamics have an impact on the artifact. We attempt to answer a few questions within this context with the generic theme of developer interaction and collaboration, and in the same time within the range/interest of Empirical Software Engineering field.

Acknowledgments

I would like to thank Professor Vladimir Filkov for believing in me, without whom I would not be able to continue my PhD research. His guidance, and more importantly his trust in my capabilities made this possible. He was and will be a mentor, a colleague, and a friend.

I would also like to thank Professor Raissa D'Souza. Her frank and direct feedback guided my research from pitfalls.

I would like to thank all the (former and current) members in our research group (DECAL) in the Department of Computer Science at University of California in Davis, Specially Prof. Premkumar Devanbu, Dr. Daryl Posnett, Dr. Bogdan Vasilescu, Prof. Qi Xuan, and Prof. Baishakhi Ray for the inspiration, valuable discussion about the ideas and technical details presented within this manuscript and beyond.

I gratefully acknowledge support from the Air Force Office of Scientific Research, award FA955-11-1-0246, which supported my research throughout most of my PhD studies. My research internship at UCR was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA)

Preface

All of the chapters (except the introduction chapter) in this dissertation have been published (or are under review to be published) as conference or journal papers. The following is the list of publications arising from this work:

Chapter 2: **Gharehyazie M.**, Zhou B., Neamtiu I., (2016) "Expertise and Behavior of Unix Command Line Users: An Exploratory Study", in *The Ninth International Conference on Advances in Computer-Human Interactions*

Chapter 3: **Gharehyazie M.**, Posnett D., Vasilescu B., Filkov V., (2014) "Developer Initiation and Social Interactions in OSS: A Case Study of the Apache Software Foundation", *Empirical Software Engineering*.

Chapter 3: **Gharehyazie M.**, Posnett D., Filkov V., (2013) "Social Activities Rival Patch Submission for Prediction of Developer Initiation in OSS Projects", in *Proceedings of the 29th IEEE International Conference on Software Maintenance*.

Chapter 4: Xuan Q., **Gharehyazie M.**, Devanbu P., Filkov V., (2012) "Measuring the Effect of Social Communications on Individual Working Rhythms: A Case Study of Open Source Software", in *Proceedings of the Social Informatics Conference*.

Chapter 5: **Gharehyazie M.**, Filkov V., (2016) "Tracing Distributed Collaborative Development in Apache Software Foundation Projects", *Empirical Software Engineering*. (under review)

Chapter 6: **Gharehyazie M.**, Ray B., Devanbu P., Filkov V., (2016) "From Here, There, and Everywhere: Cross-Project Cloning in GitHub;", in *ACM SIGSOFT International Symposium on the Foundations of Software Engineering* (under review).

CHAPTER 1

Introduction

1.1. Why Study Open Source Software?

The Open Source movement in software (a movement that is not new in some other domains) has definitely changed the world of Software, and along with it our lives. Not only have Open Source Software enabled us by providing good quality software loyalty-free, but they have also made Software Enterprise Giants such as Microsoft a run for their money, making them more agile and competent. And the winner in this fight is of course, the consumer. But Open Source is not just about free software (or other goods in different contexts), it is an idea that can open doors to whole new possibilities. Just search “open source” or “open world” in TED.com and look at the amazing talks regarding openness and open source possibilities.

But what is an Open Source Software (OSS)? Based on the Open Source Initiative, an OSS means a software that its distribution terms follow 10 criteria [**1**]. To summarize the important points, the software and its source code have to be distributed free and royalty free. We have definitely used one or more of these software either directly like R, Mozilla Firefox or Google Chromium, Ubuntu, Android OS, LibreOffice and *etc.*, or indirectly such as MySQL, PostgreSQL, Debian and Apache that are used to provide us with online web services from both non-profit and for-profit organizations.

Being an important part of almost everyone’s daily lives, OSS are interesting phenomenon to study. Even commercial software giants such as Microsoft, IBM and Apple have come to embrace OSS. The Openness of these Software (not only in the software and its source code, but also the development history, code repositories, communication logs and *etc.*) allows a unique opportunity to research them. An opportunity that is almost never available with commercial software companies. Studying OSS helps achieving a better understanding of OSS, and in turn allows us to better embrace it and utilize its power by learning how we can improve them and benefit more from them.

1.2. Why Study OSS Developers?

While OSS has been the subject of a multitude of research efforts, there are several questions that have not been fully answered yet. The most existential one being *“How do OSS exist and compete with high* *end commercial software?"* We used to think that creating something complex and complicated requires complexity, structure, organization, and coordination. That was certainly the case for a lot of things like cars for instance. In the software market, a company like Microsoft spends years and millions of dollars to develop the next Operating system and/or Office suite. How can Linux Operating Systems exist then? How do OSS manage to not only survive, but thrive and flourish? In order to answer this question, and in turn gain better understanding of OSS, we need to study OSS developers. There are two main reasons for this:

- The first major characteristic of OSS is being free, and that starts with developers who decide to give away the fruit of their labor for free and also work for free or at least seek payment through a different and less conventional channel.
- Developers are the ultimate creators of the final artifacts and studying them and their interactions sheds light on how OSS are developed.

When studying OSS developers, we need to ask and answer several smaller questions which will ultimately shed light on the bigger picture. Questions such as: Why do developers choose to contribute to OSS? How do developers communicate and organize in OSS? What are the effects of remote (vs. local) development on the software itself? Are there latent organizational structures within OSS developer communities? This is flavor of questions that we ask and attempt to answer in this manuscript.

1.3. Our Contributions

We start by attempting to extract user expertise based on user command traces in Unix systems in Chapter 2. We utilize three sources of user command line traces and infer command expertise based on several factors such as the category of the command and the frequency of its usage. We then use the implicated expertise of these commands to discuss user expertise. The arising patterns revealed many insights into user expertise and behavior, such as: a user's command length is *not* an indicator of their expertise; users activity is highest on Monday and decreases every day through Saturday, picking up on Sunday; peak command usage hours are 11 a.m., 1 p.m. and 4 p.m.; and finally development activities happen mostly in the afternoon.

Next we use contributors' socialization patterns to *predict* developer initiation in Chapter 3. For 6 different Apache Software Foundation OSS projects we compile and integrate a set of social measures of the communications network among OSS project participants and a set of technical measures, *i.e.,* OSS developers' patch submission activities. We use these sets to predict whether a project participant will become a committer, and to characterize their socialization patterns around the time of becoming committer. Our findings show that the social network metrics, in particular the amount of two-way communication a person participates in, are more significant predictors of one's likelihood to becoming a committer. Further, we find that this is true to the extent that other predictors, *e.g.,* patch submission info, need not be included in the models. Interestingly, we find that on average, for each project, one's level of socialization ramps up before the time of becoming a committer. After obtaining committer status, their social behavior is more individualized, falling into few distinct modes of behavior. Finally, we find that it is easier to become a developer earlier in the projects life cycle than it is later as the project matures.

In Chapter 4 we study how communication among pair of developers affects their technical work. We define the notion of a working rhythm as the average time spent on a commit task and we study the correlation between working rhythm and communication frequency. We find that the developers with higher social status, represented by the nodes with larger number of outgoing or incoming links, always have faster working rhythms and thus contribute more per unit time to the projects. We also study the dependency between work (committing) and talk (communication) activities, in particular the effect of their interleaving. These findings suggest that frequent communication before and after committing activities is essential for effective software development in distributed systems. We build upon this work in Chapter 5 by implicating collaboration in OSS using these talk-work patterns and define and use this information to study teams and co-development in OSS. Here we characterize co-development in larger groups, specifically with respect to developer focus and productivity. We develop a methodology for capturing distributed collaboration based on synchronized commit activities among multiple developers, and apply it to data from 26 OSS projects from the Apache Software Foundation. We use both quantitative and qualitative methods, including a developer survey to verify our methodology and results. We find that while in distributed collaborative groups, developers' behavior is different than when programming alone, *e.g.,* high developer focus on specific code packages associates with lower team participation, while packages with higher ownership get less attention from groups than from individuals. Finally, we show that productivity effort during co-development is more often lower for developers while they co-develop in groups.

Finally we move to higher level where we study code clones across projects in the GitHub ecosystem in Chapter 6. Code reuse, in general, has many well-known benefits on code quality, coding efficiency, and maintenance. Hence, programmers are happy to share their high-quality code; they also reuse relevant code from wherever they can get it. Open source software development on social programming platforms like

GitHub is taking code reuse one step further, enabling code search and reuse across different projects. Removing the project borders facilitates more efficient code foraging, and consequently faster programming. Code clones are commonly used as operational evidence of code reuse. To understand the extent of code foraging and reuse, we conduct in depth study of cross-project cloning in GitHub ecosystem. We identified similar code fragments across multiple projects, and investigate their prevalence and characteristics using statistical and network science approaches, and with multiple case studies. We find that cross-project cloning is prevalent in GitHub, ranging from cloning few lines of code to whole project repositories. Some of the projects serve as the popular source of clones, and others are code sinks. Moreover, we find that ecosystem cloning follows an onion model: most clones come from the same project, then from projects in the same application domain, and finally from projects in different domains.

All the work discussed above which are presented in more detail in the following chapters, show the potentials and promises of studying the human and social dynamics of software development and its effects on software engineering and its community.

CHAPTER 2

Expertise and Behavior of Unix Command Line Users: an Exploratory Study

2.1. Introduction

Unix, Unix-like, and Linux operating systems dominate the market in segments where human-computer interaction is centered around command-line; specifically, the market share of these operating systems, in January 2016, was 66.9% (server), 100% (supercomputer) and 100% (mainframe) [**2**]. Understanding Unix command usage and the behavior of Unix users has many applications: repetitive sequences of commands can be transformed into scripts for correctness and ease of use; temporal patterns can expose busy and idle periods hence allowing capacity to be scaled accordingly; and noticing that a user U's commands or temporal access patterns are markedly different compared to U's prior commands and usage patterns can indicate a masquerade attack [**3**] (*i.e.,* U's account has been compromised and is being used by a malicious user M). Quantitative measures of user expertise can serve as a base for comparing users (e.g., who are the most competent or efficient users?), inferring user roles (e.g., student vs. seasoned developer vs. system administrator) or assessing how users learn.

Few studies have focused on understanding user behavior based on command line usage. Rather, prior studies' focus has been on: masquerade detection (malicious users taking control of a legitimate user's account) [**4**] [**5**] [**6**] [**7**] [**8**] [**9**] [**10**]; user session characterization [**11**] in terms of commands per session, errors encountered and directories explored; or predicting high-level user actions [**12**]. In Section 2.2, we provide a detailed comparison with related work. However, none of these previous studies have attempted to quantify user or command expertise and study temporal patterns associated with users/expertise.

We start by describing the datasets and the approach we have used to identify commands and their arguments (Section 2.3). Identifying individual commands is nontrivial due to several reasons, such as the myriad ways in which Unix commands can be chained or used as arguments for other commands.

In Section 2.4, we discuss the methodology we have used for quantifying expertise and assigning expertise values to commands and users. First, we assign an *expertise* value to each command — the higher the value, the higher the probability that the command is used by more advanced users, or requires more advanced knowledge. Next, using command expertise, we assign expertise values to entire command lines. We then group commands into categories such as file management, editor, and compiler. Based on these metrics, we introduce metrics for users expertise: *user category breadth* to quantify user expertise in terms of the number of different categories employed by that user, as well separating users into high- and low-expertise groups.

In Section 2.5, we present our findings. We first characterize commands in each dataset; we found that file management is the principal activity, with `cd` and `ls` accounting for substantial percentages of user commands. We found that the most common command length is *two*; that command line length is not necessarily an indicator of expertise, and that commands/users permit a natural separation into high- and low-expertise commands/users.

We also analyzed temporal patterns in one of the datasets (the only dataset to contain command timestamps). We found that command activity tends to decrease from Monday until Saturday, and increase slightly on Sunday; that peak command usage hours are 11 a.m., 1 p.m. and 4 p.m.; and that development activities (use of editors and compilers) peaks in the afternoon.

2.2. Related Work

Schonlau et al. [**4**] have used various statistical methods to detect masquerade attacks by finding "bad data" (attacker's commands) inside sequences of "good data" (benign commands issued by a legitimate user). Other studies have similarly used machine learning or grammars for masquerade detection [**5**] [**6**] [**7**] [**8**] [**9**] [**10**]. Greenberg [**11**] has collected traces of 168 users via a modified `csh` shell; they partitioned the users into four non-overlapping categories (novice programmers, experienced programmers, computer scientists and non-programmers) and characterized user sessions in terms of commands per session, errors encountered and directories explored. Chinchani et al. [**13**] has focused on the reverse problem of generating user models so that synthetic command traces can be generated automatically. Fitchett and Cockburn [**12**] developed a predictor model for revisitation/reuse based on user actions (commands, window switching, URL accesses).

Note that our focus is different compared with the aforementioned efforts. Rather than finding suspicious/anomalous commands for purposes of masquerade detection, we aim to answer more general questions: how can command expertise, command line expertise, and user expertise be operationalized? How

TABLE 2.1. Feature availability for each dataset.

Datasets	Mahajan	Greenberg	Schonlau
Users	45	168	50
Commands per user	709[*]	1,441[*]	15,000
Command timestamp	✓		
Session start/end		✓	
Command arguments	✓	✓	
Command chaining	✓	✓	
User Group		✓	

[*] Median number of commands per user.

can users and commands be grouped into categories that generalize and are stable across datasets? What are the temporal patterns associated with commands, command categories, and users, e.g., when (time-of-day/day-of-week) does development happen, as opposed to editing, and when are experts active compared to novice users?

2.3. Datasets

Our analysis is based upon three main datasets — collected by other researchers [**10**] [**4**] [**11**] — totaling 263 users and 1,023,993 commands. Table 2.1 provides an overview of the datasets and the features they include: each dataset consists of real commands collected from actual usage — ranging from 709 up to 15,000 commands per user. The dataset attributes vary: while Mahajan's set contains a timestamp for each command, the other sets do not; conversely, Greenberg's set has session start and end markers while the other two do not. Finally, Mahajan and Greenberg's sets contains command arguments, including the chaining of multiple commands on the same line (*e.g.,* via the pipe operator), while Schonlau's only contains a command prefix (first 8 characters).

We now describe each dataset, its features, and the methodology we used to extract characteristics from that particular set.

2.3.0.1. *Mahajan.* The richest, most detailed data was gathered by Mahajan [**10**]: command traces for 45 users. Each command has a timestamp and the entire command line, *e.g.,* including complex pipes, was captured. To parse each line, we first separate the command portion into sections of commands by pipes (the '|' character) and logical commands such as '&&'. After separating a line into sections, each section contains a command, along with zero or more parameters. For example, the following line will be separated into 3 sections:

```
ls -l | grep key | less
```

Backquotes or “backticks” are commands that are executed before the rest of a line, and their result is “pasted” at their position in the line. For example:

```
vim notes.`date +%F`
```

Hence we look for backquotes in each section and treat it as a separate section; we find the command and its parameters (as described below) and we treat the whole section as an extra parameter for the section which contained the backquotes.

To count the number of parameters for each command, we separate each section by space and redirection characters (‘>’, ‘>>’, ‘<’, and ‘<<’). As mentioned above, each section is split based on space or redirection characters. Redirection to/from a file is also considered a parameter. In some rare cases there are two commands per section such as `sudo apt-get install blah` which contains two commands (‘`sudo`’ with 0 parameters and ‘`apt-get`’ with 2 parameters). Only two commands were found that used such a feature: ‘`sudo`’ and ‘`time`’.

To conclude, in the aforementioned example, we detect 3 commands: ‘`ls`’ with 1 parameter, ‘`grep`’ with 1 parameter, and ‘`less`’ with no parameter. Notice that all commands after the first pipe have one more parameter than immediately visible, and that is because the other parameter has been piped.

Finally, we proceeded to identifying *sessions*, *i.e.,* start and end of time intervals when users started the command line interaction. While Mahajan’s dataset is very rich in most aspects, it does not explicitly record session information, so to identify sessions we computed the time intervals between consecutive commands, plotted their distribution, and visually identified cut-off points hence session begin/end.

2.3.0.2. *Greenberg.* This dataset consists of 168 Unix users [**11**]. Like Mahajan’s dataset, it contains command parameters, and complex command lines, but it does not contain timestamps like Mahajan’s. It does however contain session delimiters. Another feature of this dataset is that it categorizes users into 4 categories based on their expertise. In detail, there are 52 computer scientists, 36 experienced programmers, 55 novice programmers and 25 non-programmers respectively. This makes it particularly valuable in evaluating our expertise extraction methods. Next, we show an example of the information contained in Greenberg’s dataset (the dataset contains more information which is irrelevant to this study and has been removed from the example for clarity).

```
S Wed Feb 18 16:37:25 1987
E Wed Feb 18 16:56:22 1987

C date
C nroff terry.abs | enscript
C p audio.mail

S Fri Feb 20 12:41:39 1987
E Fri Feb 20 14:24:06 1987

C mail
C rlogin sun-fsa
C rlogin sun-e
C rlogin sun-b
...
```

2.3.0.3. *Schonlau.* Although this dataset is large, containing 15,000 command traces per user for 50 users, it is limited in three ways. First, the parameters for each command are omitted; this leads to missing complex pipes and chains of commands, which can be used to assign user expertise. Second, due to the method used for gathering data, system commands, *e.g.,* commands invoked from C programs via `system()` or `exec()` were included. Finally, the command traces do not have timestamps, so processing any temporal information such as number and duration of each user session is impossible. On the flip side, parsing this dataset is trivial, since each line contains a simple command, with no parameters, no loops, no pipes, etc.

2.4. Expertise

We used the following approach for developing a uniform notion of expertise across the three datasets: we first assign an expertise value $E(C)$ to each command C, then for each command line L, including lines that consist of the chainings commands $C_1, \ldots, C_n$, we assign an expertise coefficient $lCoef(L)$ based on constituent command(s) expertise as well as the command chaining information.

Command Categories. We group commands into *categories*, *e.g.,* `vim` or `emacs` are categorized as *editor* commands, while `ping` or `traceroute` are *network* commands, and so on. In total we have defined

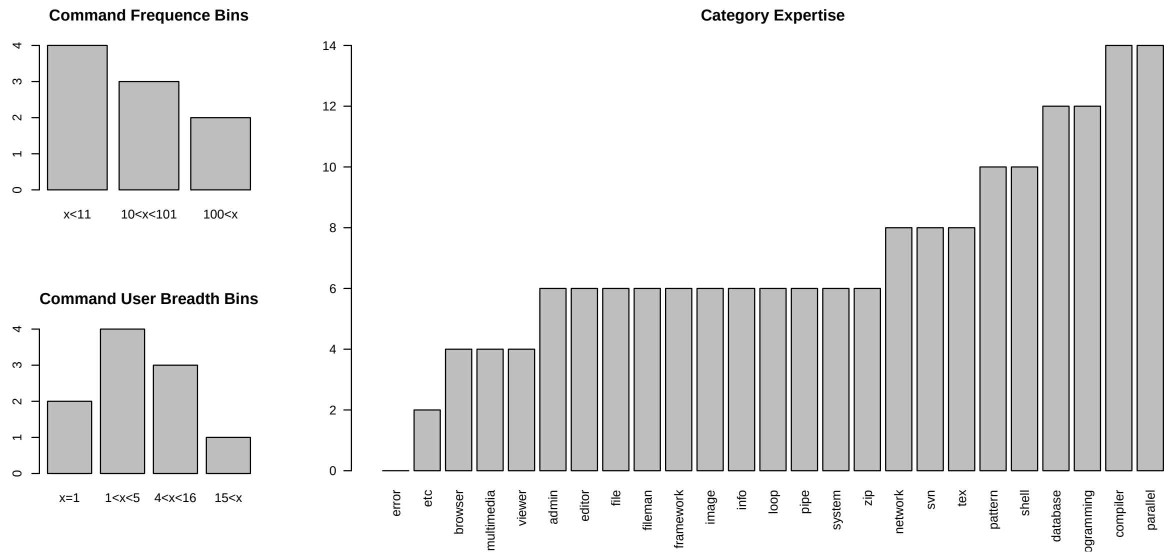


FIGURE 2.1. Expertise assigned to bins of a) command frequency (top left); b) user breadth (top right); and c) command category (bottom) for the Mahajan dataset.

25 categories. The number of categories varies little across datasets: 24 for Mahajan, 23 for Schonlau and 18 for Greenberg (Table 2.2) which indicates that our category definitions are stable.

2.4.1. Command and Line Expertise.

2.4.1.1. *Assigning Expertise to Commands.* Due to inherent differences among the three datasets, they could not be merged into one dataset, and thus assigning expertise was performed separately on each of them. We first measure the number of users that have used each command (U), along with the number of times each command was used ($Freq$). We also manually assign commands to categories based on type of application.

We group U into 4 bins and $Freq$ into 3 and assign an expertise value to each bin (U_{exp} and $Freq_{exp}$) as shown on the left of Figure 2.1. A level of expertise is assigned to each category (C_{exp}) as shown on the bottom of Figure 2.1, *e.g.,* browser commands have expertise value 4, network and svn (version control) have value 10, while compiler commands have expertise value 14. Then, for each command we assign an intra-category expertise adjustment (I_{exp}), as explained next.[1] The reasoning behind the expertise assignments is:

- For user breadth U_{exp}, the more people utilizing a command, the less likely it is to require high expertise. At the same time, if a command is used only by a single person, there is a high chance

[1]Expertise values were assigned based on consensus among authors.

that it is highly personal, *e.g.,* scripts. We believe the expertise associated with this bin should be lowered, but not lower than the bin with highest number of users.

- For $Freq_{exp}$, a command that is used less frequently is likely associated with higher expertise.
- A category of commands C_{exp} is the highest indicator of expertise, *e.g.,* a compiler command is much more complicated than a browser command.
- Within a category, there are certain commands that require a higher expertise to be used; these commands are given extra expertise points in intra-category expertise I_{exp}.

Therefore, we use the following formula for assigning expertise to each command C_i:

$$Exp(C_i) = U_{exp}(C_i) + Freq_{exp}(C_i) + C_{exp}(C_i) + I_{exp}(C_i) \tag{2.1}$$

While the most influential element of each command's expertise is the category it belongs to, other elements can boost or hinder its expertise and thus provide a wide range of values. For example, in Schonlau's dataset, expertise values range from 6 to 22.

2.4.1.2. *Assigning Expertise to Lines.* For the dataset of Schonlau et al., each line's expertise is the same as the command in that line. But for Mahajan and Greenberg datasets, we need a measure to combine multiple commands' expertise into a single value. Adding up all the expertise is not a great idea since it can highly inflate the expertise values, and it is certainly misleading, since someone who uses 10 commands in a single line, does not necessarily possess 10x greater expertise in comparison to another person who uses the same commands on separate consecutive lines. We believe that the former's expertise is merely a fraction more than the latter's and that is due to their ability to combine multiple commands in a single line and perform complicated tasks. To this end, we have developed the following line expertise computation scheme:

$$lCoef(l_i) = \begin{cases} 1 + \frac{min(|Commands(l_i)|,4)}{10}, & \text{if } l_i \text{ contains backquotes} \\ 1 + \frac{min(|Commands(l_i)|,4)-1}{10}, & \text{otherwise} \end{cases} \tag{2.2}$$

$$Coef(C_j) = 1 + \frac{min(|Params(C_j)|,4) - 1}{10} \tag{2.3}$$

$$Expertise(l_i) = lCoef(l_i) \times \max_{C_j \in Commands(l_i)} Exp(C_j) \times Coef(C_j) \tag{2.4}$$

Here, l_i is any given line, C_j is a member of the set of commands within l_i, denoted by $Commands(l_i)$ and $Params(C_j)$ gives the set of parameters for C_j. $lCoef(l_i)$ is an expertise coefficient for line l_i which gives a 10% boost to l_i's expertise for each extra command in l_i, up to 4 commands; it also gives an extra 10% boost if backquotes are used in l_i. $Coef(C_j)$ is a command coefficient which gives a 10% boost to C_j's expertise for each extra parameter provided to C_j, up to 4 parameters.

2.4.1.3. *High- and Low-Expertise Commands.* To provide a straightforward binary separation into "easy" and "advanced" commands, we use the command expertise distributions to divide all commands into low and high expertise groups. These in turn will be used to support separating users into low- and high-expertise groups. We present the results in Section 2.5.1.3.

2.4.2. User Expertise. We now provide definitions of user expertise.

2.4.2.1. *User Category Breadth.* User Category Breadth (UCB), *i.e.,* the number of different categories they actively use, is also an indicator of expertise. To measure UCB, we identify the category of commands for each user, and apply a minimum number of commands threshold to weed out those cases where accidental or extremely infrequent use of commands would count toward category use. Specifically, we used 1.5% of the average number of commands for each user in each dataset to define the aforementioned threshold — this translates to 20 commands in Mahajan's dataset and 250 commands in Schonlau's, *i.e.,* to count a category C toward a user U's UCB, U has to have used at least 20 and 250 commands in C, respectively.

2.4.2.2. *High- and Low-Expertise Users.* To distinguish different users, we divide them into low and high expertise groups based on high- and low- command line expertise mentioned in Equation (2.2). We found a clear delineation between the high- and low- expertise sets; we present the results in Section 2.5.2.2.

2.5. Results and Discussion

We now proceed to discuss our findings.

2.5.1. Command Characteristics.

TABLE 2.2. Summary of command characteristics.

	Mahajan	**Greenberg**	**Schonlau**
Total Commands	56,261	313,169	750,000
Unique Commands	1,218	4,117	856
Categories	24	18	23

TABLE 2.3. Top-20 commands for each dataset.

Rank	**Mahajan**		**Greenberg**		**Schonlau**	
	Command	%	Command	%	Command	%
1	`cd`	15.3	`ls`	12.3	`sh`	8.7
2	`ls`	15.0	`cd`	8.8	`cat`	4.3
3	`git`	3.7	`pix`	6.2	`netscape`	4.3
4	`vim`	3.3	`umacs`	4.9	`generic`	4.1
5	`sudo`	3.1	`e`	4.2	`ls`	4.0
6	`grep`	2.8	`rm`	3.1	`popper`	3.3
7	`vi`	2.6	`fg`	3.1	`sendmail`	2.8
8	`gvim`	2.3	`emacs`	3.0	`date`	2.7
9	`ssh`	2.2	`more`	2.8	`rm`	2.3
10	`mm`	2.2	`lpq`	1.9	`sed`	2.1
11	`rm`	2.0	`mail`	1.8	`nawk`	2.0
12	`java`	1.9	`lpr`	1.8	`expr`	1.9
13	`perl`	1.6	`cat`	1.8	`tcsh`	1.8
14	`javac`	1.5	`cp`	1.4	`grep`	1.7
15	`find`	1.4	`ps`	1.3	`tcpostio`	1.4
16	`clear`	1.2	`nroff`	1.2	`uname`	1.4
17	`mount`	1.1	`who`	1.1	`ln`	1.3
18	`cp`	1.1	`make`	1.0	`hostname`	1.3
19	`cat`	0.9	`fred`	0.9	`gcc`	1.3
20	`exit`	0.8	`u`	0.8	`true`	1.3

2.5.1.1. *Command Distribution.* Table 2.2 shows the summary of each dataset. Although the Schonlau dataset has the most commands, it has the fewest *unique* commands. Note how, despite differences in datasets (*e.g.,* provenance, year of collection, Unix system used) they have similar numbers of command categories, which indicates that our category definitions are quite stable across different Unix user populations.

Table 2.3 shows the top-20 most used commands and their percentage in each dataset. File management commands (`ls`, `cd`, `cp`, and `rm`) are the most popular by far, and they dominate the Mahajan and Greenberg datasets (more than 20% of their commands fall into this category). The Schonlau dataset is more evenly distributed, but file management is popular there as well.

TABLE 2.4. Top-20 commands of each category in Greenberg dataset.

Rank	Scientist		Experienced		Novice		Non-Programmer	
	Command	%	Command	%	Command	%	Command	%
1	ls	14.6	cd	12.0	pix	24.4	ls	15.8
2	cd	10.5	ls	11.8	umacs	19.7	emacs	10.6
3	e	5.2	e	5.8	ls	7.8	nroff	9.1
4	fg	4.4	fg	4.6	rm	3.4	cd	8.5
5	rm	3.0	more	4.1	u	3.1	e	5.3
6	mail	2.8	rm	2.7	cd	2.9	rm	4.0
7	emacs	2.4	make	2.7	cat	2.7	ee	3.8
8	lpq	2.2	emacs	2.5	more	2.6	more	3.4
9	more	2.1	lpr	1.9	script	2.4	lpr	3.0
10	ps	1.8	l	1.9	lpr	2.4	hpr	2.8
11	f	1.6	cat	1.8	lpq	2.0	ptroff	2.2
12	cat	1.6	ada	1.8	cp	2.0	lpq	1.9
13	who	1.5	examples_vax	1.7	emacs	1.8	ps	1.8
14	mv	1.1	cp	1.5	pi	1.5	cp	1.4
15	lpr	1.1	a.out	1.3	p	1.2	tbl	1.4
16	man	1.1	rwho	1.3	mail	1.0	w	1.3
17	rlogin	1.0	mail	1.2	fred	1.0	col	1.2
18	cp	1.0	lpq	1.2	logout	0.8	mail	1.2
19	page	0.9	bye	1.2	pdpas	0.7	rr	1.1
20	fred	0.9	ps	1.2	man	0.6	spell	1.1

TABLE 2.5. Top-5 categories for each dataset.

Rank	Mahajan		Greenberg		Schonlau	
	Category	%	Category	%	Category	%
1	fileman	39.0	fileman	28.3	pattern	13.7
2	etc.	8.4	editor	14.7	framework	12.2
3	editor	8.2	info	13.1	etc.	12.1
4	pattern	5.8	etc.	10.7	system	11.8
5	compiler	5.1	compiler	9.2	fileman	10.1

Furthermore, we investigate whether commands differ between user groups. Table 2.4 shows the top-20 most used commands and their percentage of user groups in the Greenberg dataset. For groups computer scientist, experienced programmer and non programmer, similar to the whole dataset, file management commands contribute the most, but for the novice programmer group, the compiler-related command `pix` and (Pascal interpreter and executor) and editor command `umacs` are the most used commands.

Table 2.5 shows top-5 most used categories and their percentage in each dataset. The observations are similar to the previous observations on commands: file management is prevalent, with Mahajan and Greenberg's datasets having a higher concentration of such commands compared to Schonlau's.

TABLE 2.6. Top-5 categories for user groups of the Greenberg dataset.

Rank	**Scientist**		**Experienced**		**Novice**		**Non-Programmer**	
	Category	%	Category	%	Category	%	Category	%
1	fileman	32.3	fileman	31.7	compiler	27.4	fileman	31.2
2	info	14.4	etc.	13.2	editor	23.3	editor	21.1
3	etc.	11.8	info	12.6	fileman	17.2	info	13.1
4	editor	10.6	editor	10.7	info	11.6	Tex	11.8
5	system	9.1	system	8.4	system	7.4	etc.	11.1

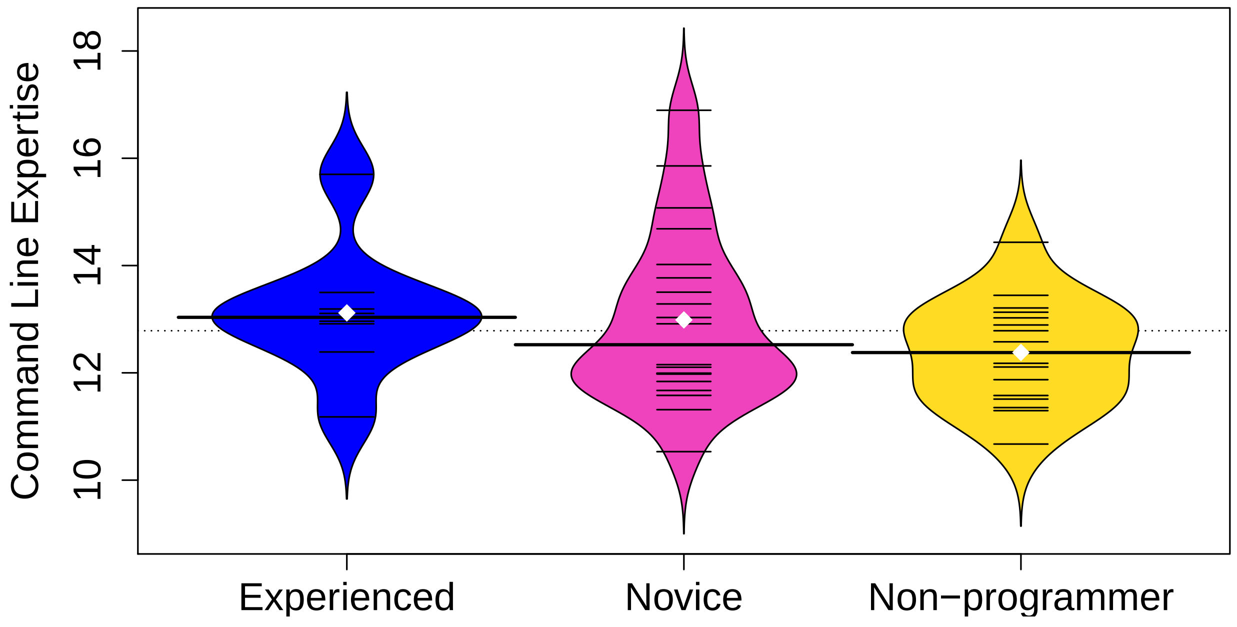


FIGURE 2.2. Beanplot of average command line expertise of each user group in Mahajan's dataset.

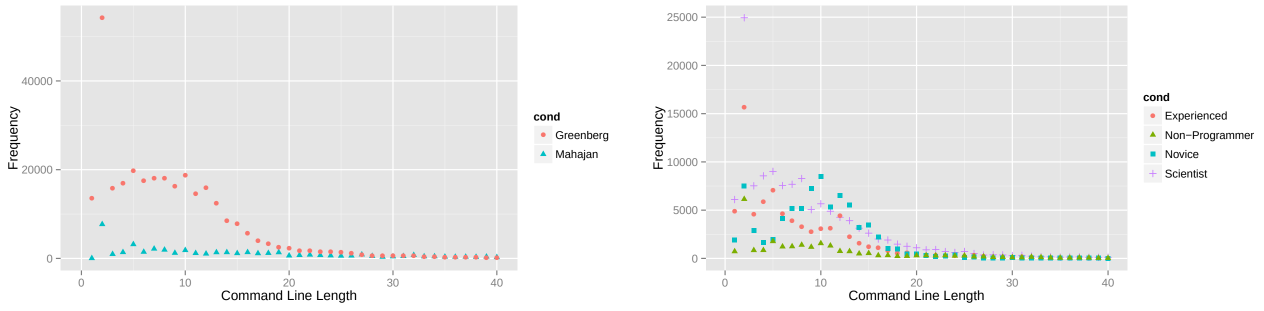


FIGURE 2.3. Distribution of command line length in Mahajan and Greenberg datasets (left) and the four user groups of the Greenberg dataset (right).

The Greenberg dataset comes with an assignment of users into groups: computer scientists, experienced programmers, novice programmers, and non-programmers [**11**]. Therefore, for this dataset alone, we have investigated category distribution for each group. According to Table 2.6, we found that computer scientists and experienced programmers' groups have similar top categories, *e.g.,* file management is the most frequent used commands. However, the novice programmer group used compiler and editor commands most often, whereas non-programmers, as expected, used file management, editor, help (info) and TeX (text processing) commands most often.

2.5.1.2. *Command Line Expertise.* We now attempt to illustrate how command line expertise differs among user groups. We classified the Mahajan dataset into 3 groups: experienced programmer, novice programmer and non programmer; we performed this classification manually by sampling each user's commands and evaluating the categories and expertise of the sampled commands. Figure 2.2 shows the bean-plot of average command line expertise of each group in Mahajan. The shape of the bean-plot is the entire density distribution, the short horizontal lines represent each data point, the longer thick lines are the medians, and the white diamond points are the means. While the results show that the peak density of expertise is higher for the experienced users and then for novice programmers, a *Mann-Whitney U test* shows that the differences are not statistically significant, which very well may be due to our small sample size (a total of 45 users). So while this result suggests some merit to our approach, we cannot use it to verify our methodology.

We use command line length as a metric to check the difference between the Mahajan and Greenberg datasets, as well as the difference between user groups in Greenberg. According to the left side of Figure 2.3, command line length 2 is the most frequent pattern for both Mahajan and Greenberg which is expected, as commands `ls` and `cd` have been used most often. For scientist and experienced programmers of Greenberg dataset, we found a similar pattern. But for novice programmers, command line length 10 is the most used, while command line length 9 has similar frequency with command line length 2. Upon investigation, we found that since novice programmers were learning how to program, they used commands `umacs` and `pix` (with corresponding arguments to reach line lengths 9 and 10) more often, which result in the different trend compared to scientists and experienced programmers.

2.5.1.3. *High- and Low-Expertise Commands.* Finally, we investigate the distribution of expertise for each dataset; in Figure 2.4, we show the result. We set the median of expertise values (which empirically is 11 across all three datasets) as the threshold to separate *low* from *high* command line expertise. Note that, since the same value, 11, emerges as a natural threshold in all three distributions, we gain confidence in the stability of our command expertise metrics.

2.5.2. User Characteristics.

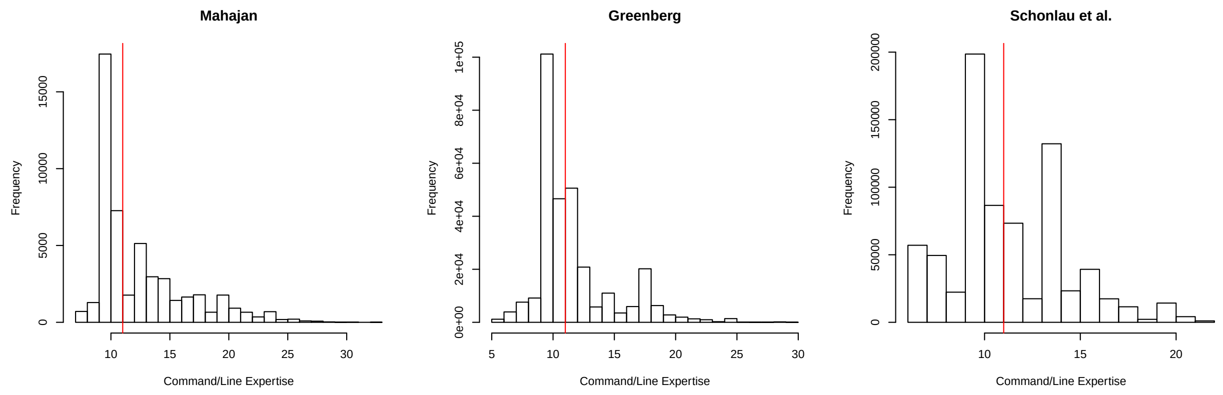


FIGURE 2.4. Distribution of expertise in commands. The red line indicates the median, which separates *low* and *high* expertise commands.

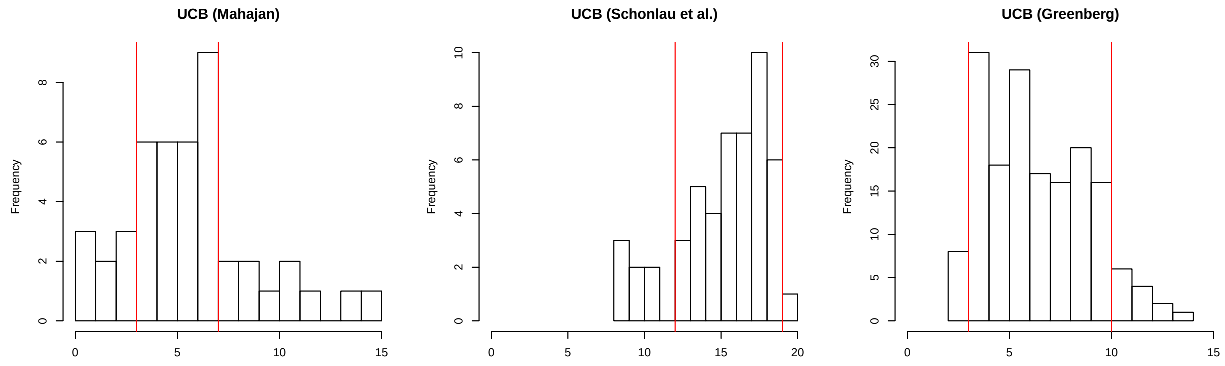


FIGURE 2.5. Histogram of number of categories of commands each user is active on. Being active in a category requires using one or more commands from that category at least 20 times.

2.5.2.1. *User Category Breadth.* We analyzed Mahajan's dataset as described above for UCB classes. The results are shown in Figure 2.5 (left). We observe 3 major patterns. First, there is a group of people who use 3 or fewer categories of commands. Second, there is a large group of people who use between 4 and 8 categories, and finally the group of people who use more than 8 categories. We name these classes *low*, *medium*, and *high* breadth classes respectively. In Mahajan's dataset, we found that 6 users are a member of low-breadth class, 10 are a member of high-breadth class, which leave the rest (29 users) in the medium-breadth class.

For Schonlau's dataset the pattern occurs again, but at different points. Here the *low* breadth class spans up to 12 categories and consists of 7 users. The *high* breadth class starts from 19 categories and consists of 1 user. This leave the *medium* group with 42 people and a range of 12 to 19 categories.

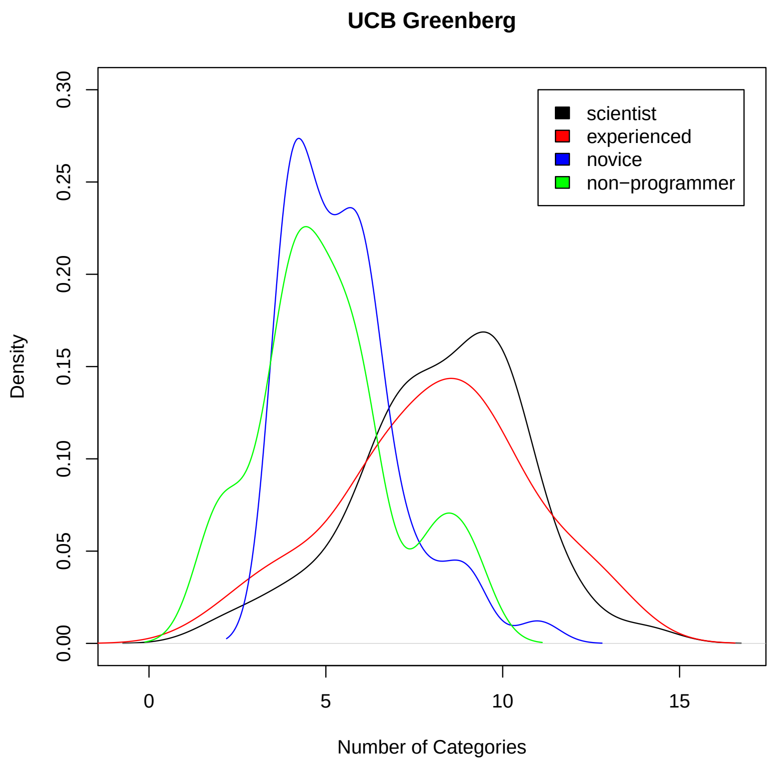


FIGURE 2.6. Kernel density function for the command category distribution of each user class in Greenberg's dataset.

For the dataset of Greenberg, we still have similar pattern with different points. The *low* breadth class spans up to 3 categories and consists of 8 users. The *high* breadth class starts from 10 categories and consists of 29 users. Then the *medium* breadth class group with 131 users and a range between 3 to 10 categories. While the values are different, the pattern of two small classes with low and high breadth and a larger class of medium breadth is still prevalent.

We also analyzed category usage in the different user groups of Greenberg's dataset. Figure 2.6 shows the kernel density function for the command category distribution of each user group. We found that novice programmers and non programmers employ fewer command categories than scientists and experienced programmers, which is intuitive.

2.5.2.2. *High- and Low-Expertise.* We now show the results of command line expertise per user. Figure 2.7 shows the result. Greenberg and Schonlau datasets have similar trends, *i.e.,* most users have only one peak value between 11 and 12; for the Mahajan dataset, we found similar trends, *i.e.,* two peak values between 11 and 13. We believe the similarities across datasets validate our choice of expertise metrics.

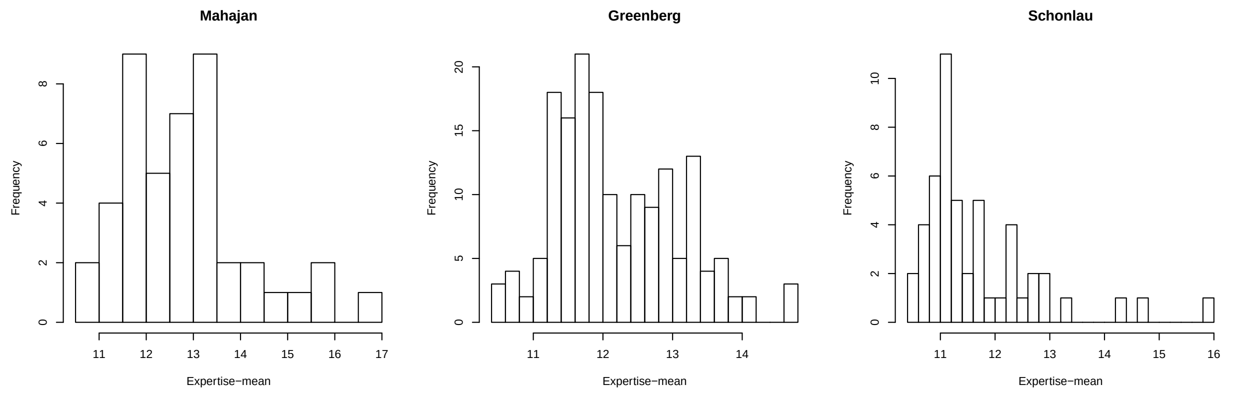


FIGURE 2.7. Histogram of mean command line expertise of each user is active on.

2.5.3. Day-of-week Command Patterns. To understand temporal patterns for command usage, we studied Mahajan's dataset (the only dataset that come with timestamps). This process is performed for all users, and then for each breadth category separately.

Figure 2.8 shows the distribution of command usage over days of week. The red lines depict high expertise commands' frequency while the blue lines indicated low expertise commands' frequency. The week starts on Monday. While the frequency tends to decrease from Monday through Sunday, we need to split users into expertise levels to better understand trends. Interestingly: (a) low- and medium-expertise users show little variation (slight decrease) as the week progresses, while for high-expertise users the trend is clear and decreasing; (b) Sunday usage is higher than Saturday usage.

2.5.4. Time-of-day Command Patterns. The distribution of command usage over hours of day is shown in Figure 2.9. The red lines depict high expertise commands' frequency while the blue lines indicated low expertise commands' frequency. There is a clear spike in activity around 11 a.m. and a rise in activity from 1 p.m. to 4 p.m. But when we split the users based on UCB, we observe different patterns for medium and high breadth classes. The Medium breadth class shows spikes of activity around 1 p.m and 3 p.m., while the high breadth class shows a spike around 11 a.m., and a relatively high activity from 1 p.m. that peaks at 4 p.m.

2.5.5. Inter-command Time. We found that users interact with the shell in sessions, *i.e.,* bursts of commands coming in rapid sequence, followed by long pauses. We studied inter-command time to get a sense as to how temporally close the commands are. Figure 2.10 illustrates this for high-expertise users in the Mahajan dataset. Note that the distribution of inter-command distances is highly skewed towards zero

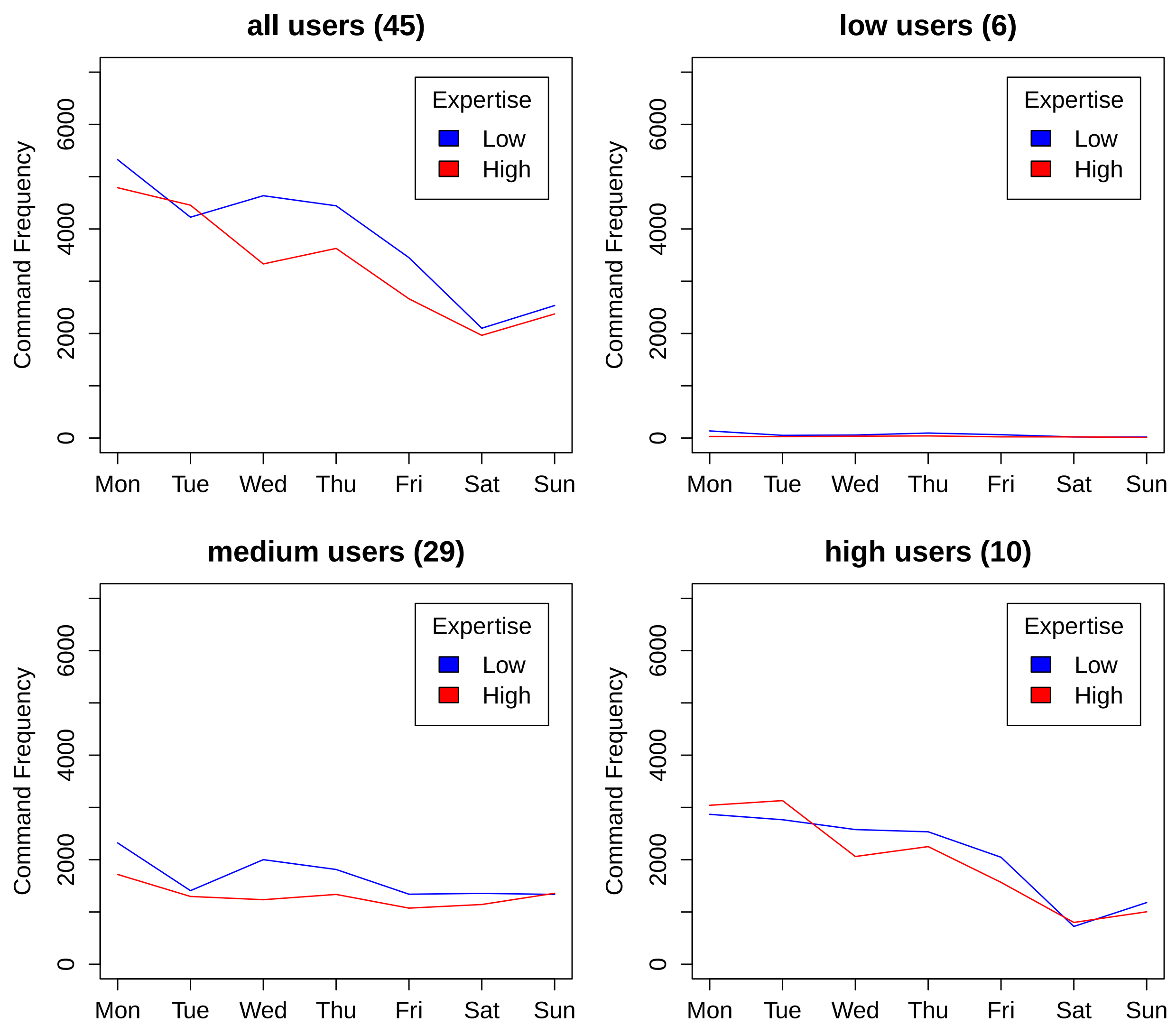


FIGURE 2.8. Frequency of low and high expertise commands used by different classes of users at different days of the week.

and has a very long tail (we trimmed both ends of the distribution — less than 30 minutes and more than 6 hours — in order to have a clearer view). As we can see in the figure, the distribution within the defined range still looks mostly exponential.

2.5.6. Category Patterns. We also analyzed the usage pattern of categories. Figure 2.11 shows category frequency for each day of week: Monday and Thursday are the days with the highest activity. Moreover, "editor" is the most popular category.

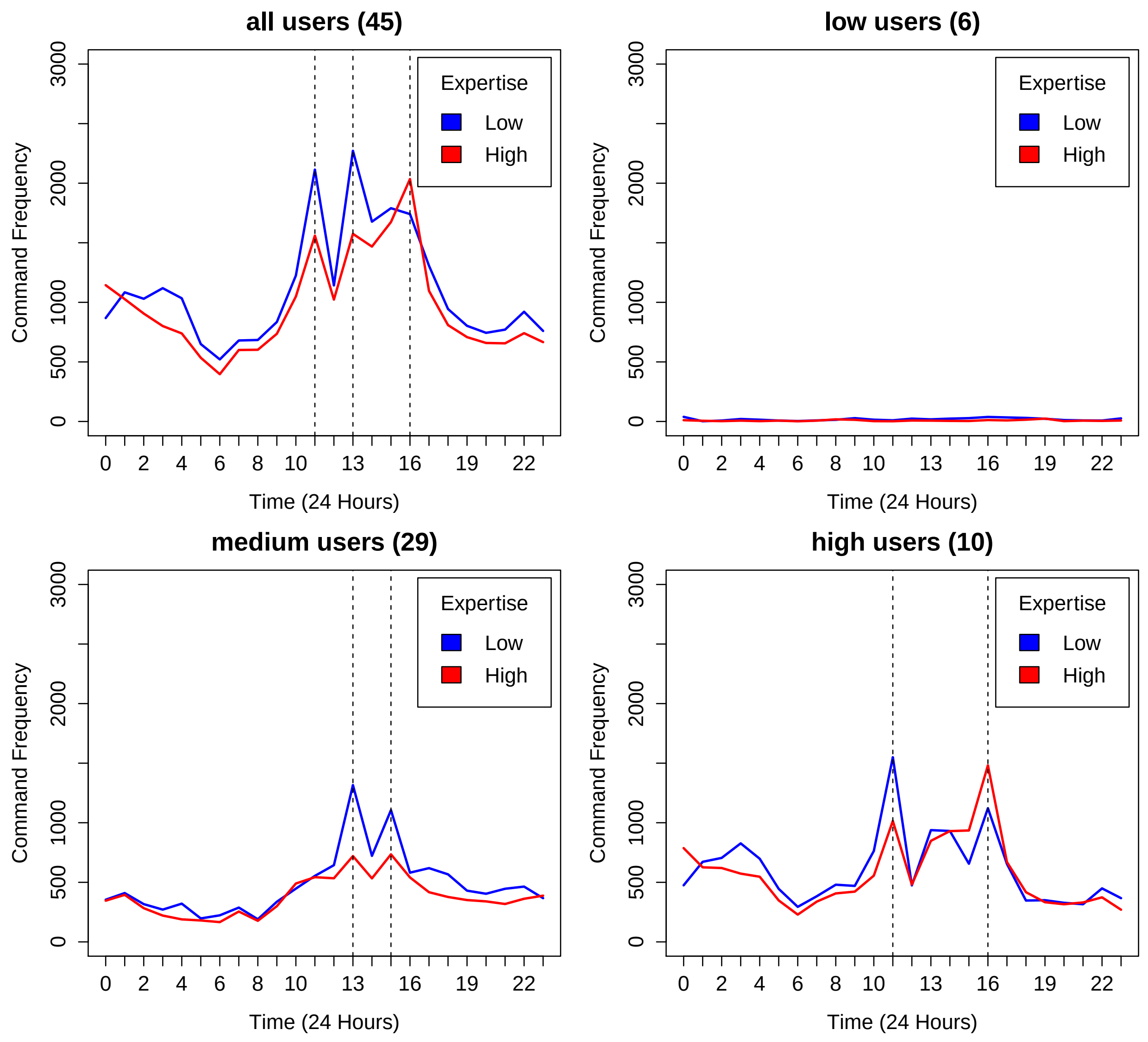


FIGURE 2.9. Frequency of low and high expertise commands used by different classes of users at different hours of the day.

Figure 2.12 shows the time-of-day results: 6 a.m. is the "quietest" time while there are usage spikes at 11 a.m. and 3 p.m.

2.6. Conclusion

We have a performed a study on three sizable datasets of Unix command usage. This is the first study to operationalize command expertise and user expertise via several metrics. Based on these metrics, we found several interesting observations on both command and user characteristics that are consistent across

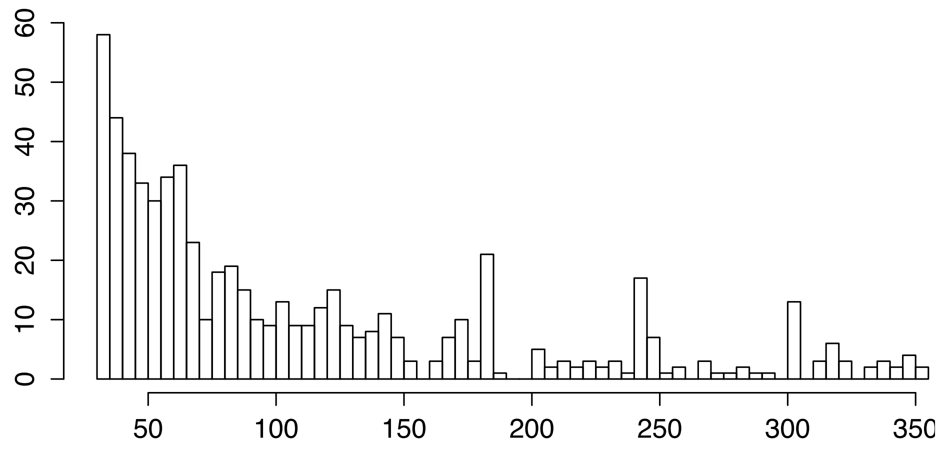


FIGURE 2.10. Histogram of temporal distance between consecutive commands for high-expertise users in the Mahajan dataset: x-axis is intercommand time in minutes, y-axis is frequency.

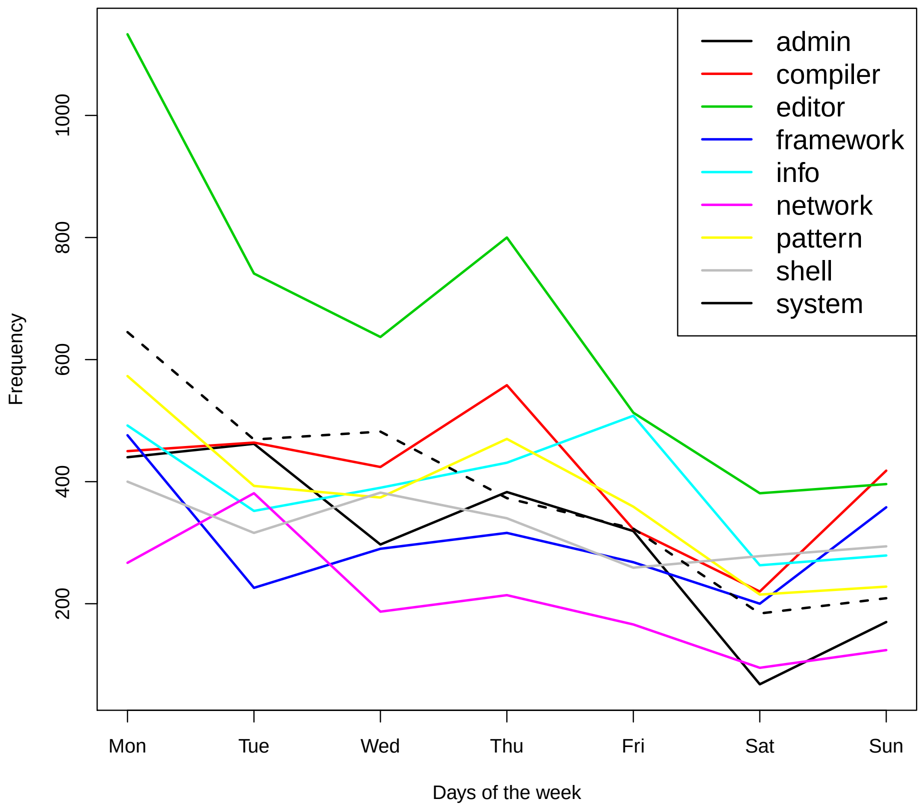


FIGURE 2.11. Frequency of several command categories used by all the users at different days of the week.

datasets, which strengthens our belief that the metrics are stable. Finally, we performed a study on one of the datasets that reveals user behavior and command usage across time of day and day of week.

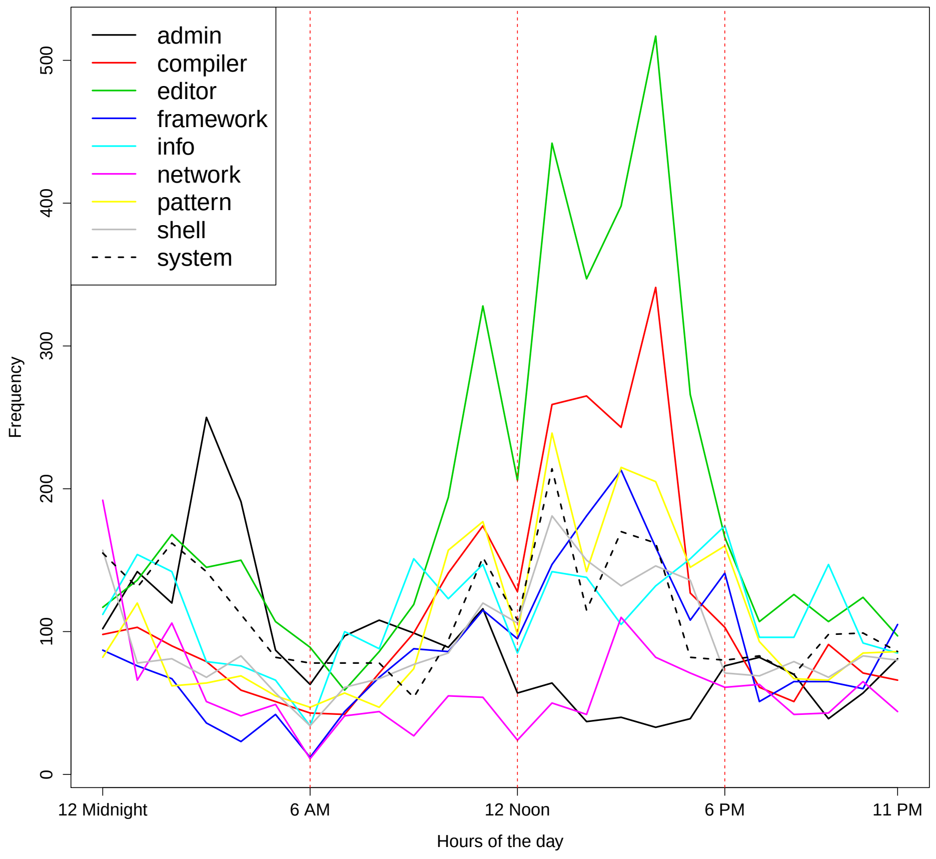


FIGURE 2.12. Frequency of several command categories used by all the users at different hours of the day.

We believe that our definitions and findings can be used in various scenarios. Our command frequency analysis can be used in real-time to identify outliers, e.g., for masquerade detection. Behavioral patterns can be used to predict Unix user's behavior which helps improve Unix users' experience, from replacing long sequences of commands with scripts to reduce the potential for errors to scaling computing capacity and scheduling support staff. Being able to quantify expertise can be useful in comparing users or assessing how users learn.

CHAPTER 3

Developer Initiation and Social Interactions in OSS: A Case Study of the Apache Software Foundation

3.1. Introduction

Open Source Software (OSS) are developed by communities of geographically- and temporally-distributed contributors ranging from professional software developers to volunteers from varied backgrounds who, despite participating in a very decentralized process, succeed to work together effectively and productively [**14**, **15**].

Well known examples of thriving OSS projects, like the Linux operating system, Apache web server, and many others, rival or even exceed the quality of commercial competitors [**16**].

Although typically lacking the organizational hierarchy characteristic of commercial settings, most OSS communities create and enforce clear cut contribution policies. The resulting community structure, commonly referred to as the "onion model" [**17–19**], comprises different contributor roles based on their level of commitment to a project's maintenance and evolution. At the core of the "onion" lie contributors with write access to a project's source code repositories (referred to as *core developers*, or committers); they have the highest level of access to the project thus can introduce changes to the code directly, but also share the greatest responsibility of delivering and evolving a viable product. The next smallest layer comprises *peripheral developers*, who typically propose smaller changes, known as *patches*, in the form of bug fixes, feature improvements, or contributions to documentation; patches are reviewed by core developers and are added to the project's source repositories at their discretion. Finally, *users* are the downstream consumers of OSS; their participation in OSS communities is mostly restricted to discussions on mailing lists or issue reports.

OSS communities are also faced with high turnover [**20**]. Therefore, to ensure a community's sustainability over time, the progressive integration of new members in all layers of the "onion" is paramount [**21**, **22**]. This process, typical of meritocratic communities [**23–25**], has received a lot of attention in the empirical software engineering research literature, where it is variously referred to as developer initiation [**26**], entering the circle of trust [**26**], migration [**27**], or immigration [**28**]. A typical trajectory to becoming a core developer is to start communicating with other project contributors and then gradually get more involved, by earning a more central position in the project's social networks and/or producing more valuable technical contributions, for example, submitting patches, or working on bug fixing activities [**21**, **28**]. Eventually, developers may reach the core of the "onion" through the recognition of their contributions.

The congruence of a newcomer's social and technical activities is crucial for successful participation in an open source project [**29**]. In a sense, a newcomer is as integral to the project as their contributions, be they communications or bug fixes. Strong patches containing working and well-tested code lead to increased trust in the developer's ability to add technical value to the project. Similarly, strong social skills signify that the developer can integrate well with the project team. The more trustworthy a developer, the more likely she is to eventually reach the core of the "onion" and achieve committer status. Generally, only those contributors who have sufficiently proven themselves through their activities become committers [**23**–**25**].

Developer initiation in OSS, thus, depends on the social and technical actions of project contributors, *e.g.,* who they talk to, the number of social links they develop with other project members, their communication patterns, patch submission activity, or bug identification and fixing activity. But to what extent do social activities and technical activities work together to increase one's chance of advancing through the ranks? And how does one's socialization behavior evolve from before to after having been recognized as core developer or committer?

In this article we revisit the issue of migration between roles in OSS projects, studying it from a social network analysis perspective. We collect data about *social* (*i.e.,* communication on mailing lists) and *technical* (*i.e.,* patch submissions and code changes) activities of developers in 6 projects from the Apache Software Foundation (ASF), arguably the most famous example of a large-scale meritocratic community [**23**–**25**]. From the data we build and compare statistical predictors for the likelihood of a newcomer to become a core developer (committer).

We find that:

- Developer initiation in OSS can be modeled very well as a social network phenomenon based solely on people's communication activities, in particular the number of two-way social links they establish, *i.e.,* messages participants *respond to*, or messages they *receive* in response to their own message.

The two-way social links are different from a one-way communication, most of which might not attract any attention or response. We find that social-links-based models exhibit better predictive ability for developer initiation than models incorporating patch submissions.

- Whether a contributor will eventually become a committer can be predicted with great accuracy from the first three months of their tenure with the project. In most cases, this is based solely on the number of their two-way social links. Furthermore, models learned on only a single month of data, containing the social links that one establishes, yield good prediction results that are within 10% of the accuracy of models learned on three months of data. In other words, as little as one month of trace data is sufficient to predict whether a contributor will reach the team core (committership).
- Contributors steadily ramp-up their social activities in each project, on average, before becoming core developers. After obtaining committer status, their social behavior is more individualized, falling into few distinct modes of behavior. In a significant number of projects, immediately after the initiation there is a notable social cooling-off period. Interestingly, and perhaps predictably, there is an apparent robustness in the communication patterns of developers with prior engagement in the ASF community, which we expound on in a case study.
- The impact of both social and technical participation declines with project age. In other words, it becomes more difficult to attain committer status as the project matures.

These findings have implications for researchers and practitioners alike. On the one hand, our work contributes to the growing body of research in empirical software engineering (*e.g.,* [**30–34**]) advocating the importance of *social* factors in the evolution of (OSS) software projects. On the other hand, OSS practitioners, be they newcomers wishing to contribute to a project, or managers interested in the sustainability of a project's community of contributors (*e.g.,* for Apache, Project Management Committees, whose role, among others, is to "further the long term development and health of the community as a whole"[1]) can use the evidence we provide to optimize their initiation or oversight processes, respectively.

The rest of the article is organized as follows. We first focus on the background behind OSS development and migration and present our research questions. Then, we describe the data and data gathering process, followed by our methods, results, and conclusion sections.

[1]`https://www.apache.org/foundation/how-it-works.html#pmc`

3.1.1. Background: The Apache Software Foundation Process. The Apache Software Foundation (ASF) community has a flat, bazaar-like [**35**] structure, in which "anyone can be a contributor"[2]. Still, different contributor roles exist, and are clearly defined[3]. Similarly to the "onion" model, in ASF one can distinguish between *users* (as in the "onion" analogy), who contribute to discussions on mailing lists or report bugs, *developers* (peripheral developers), who in addition provide patches or contribute to documentation, and *committers* (core developers), who were granted write access to the code repository and can push their changes directly. In addition, ASF committers can be elected based on merit to participate in each project's *Project Management Committee* (PMC), the entity which, as a whole, controls the project and approves active developers for committership. Finally, active committers or PMC members can be elected as *ASF members*. They are considered the "shareholders" of the foundation, with project-related as well as cross-project responsibilities and activities, such as electing the board or proposing new projects for incubation.

Apache projects as well as many other OSS projects adhere to the meritocratic governance model, in which participants gradually gain influence over a project, and consequently advance through the ranks, through the recognition of their contributions. In this article we focus on the *transition from developer to committer*, often studied in the empirical software engineering research literature [**26**–**28**]. To become a new committer, a contributor must first gain acceptance within the community and show *commitment* to the project. This can be accomplished through any number of ways—assisting users on the user list, testing code, writing documentation, bug triaging, or writing code and submitting patches for review and integration into the code base. Only when a developer has contributed sufficiently to a project, they may be nominated for committer status by an existing committer, after which voting takes place. Existing committers may then choose to grant this developer committer status, allowing her to make direct changes to the project's source code repository. Sought-after characteristics of ASF committers are "the ability to be a mentor and to work cooperatively with [one's] peers" [4] Such traits can be observed early on, during a developer's *technical* and *social* collaborative activities before obtaining committer status. Technically, submitting patches is the preferred way in which unverified code changes are communicated in many OSS projects. Submitting a bug fix patch provides a basic degree of evidence that a developer understands the software at a technical level. To have their patches accepted, developers must submit work of high quality (*e.g.,* well tested, well

[2]http://community.apache.org/contributors/
[3]http://www.apache.org/foundation/how-it-works.html#roles
[4]http://community.apache.org/contributors/

integrated with that of others) as well as advertise and argue its relevance to the project. Socially, actively contributing to mailing list discussions and offering useful suggestions and criticisms are key ways in which a new contributor can attract the attention of existing committers and gain social reputation within the community.

Hence, whether a developer will eventually become a committer is a function of all their social and technical activities within the project's ecosystem. In general, trust in a developer's technical and social skills is believed to increase with time, as a result of increased contribution and interaction with other contributors [**36**]. To illustrate this process, consider the example of John (name redacted for privacy reasons), whose first public interaction with the Apache Pluto community is on the mailing list in August 2006:

> *Hello all, I'am John from the University [...], we are developing the Prototype for the JSR 286. I hope that we can discuss the code [...] we have made and then develop new code for Pluto together [...],*

referring to his and some of his fellow student's intentions to contribute to Pluto. John gets the attention of Pluto committers and is immediately welcomed as a developer into the community, but without commit rights:

> *John, who already subscribes to this list, appears to be cooperative. He wants to work with us and contribute code to our SVN repository with the help of current Pluto committers. He already has agreed to work on two of the most important container-related Pluto 1.1 issues. I'm suggesting we create a branch based on our current Pluto 1.1 trunk. He and his group can submit their code by creating issues in this Jira branch and attaching patches to these issues. It will be up to the Pluto committers to add these contributions to SVN. In time, I hope that John can earn the right to be a committer himself [...].*

After submitting a total of 33 patches, John signals his readiness for committership in March 2007:

> *[...] Maybe we could all benefit if I can commit our patches myself. Please write me comments if we made anything wrong and how we can make it better.*

However, it seems too early for Pluto committers to trust him yet:

> *I understand your frustration. At the same time, I encourage you to continue to be active on this list. The development list is a HUGE part of the open source community, and you will only help your cause by having these types of discussions as well as airing design decisions on this list. That type of consistent communication will get you a long way - not only in getting attention from committers,*

but also in obtaining commit credentials [...] I would also recommend that your group begin to communicate exclusively on the mailing lists. This will show that you are a critical part of our community. Becoming a committer is as much (if not more) about community involvement as it is about code.

John presses forward and eventually becomes a Pluto committer in August 2007, one year after joining the mailing list.

Therefore, temporal measures of participation (both technical and social), including the number of emails sent, one's degree in the email social network, the number of committers among a developer's neighbors, the numbers of bugs reported, or the number of patches submitted, may all be indicators of the community's trust in a developer's abilities [**37**, **38**]. Different paths to committership may exist, as illustrated in Figure 3.1. There, A contributes patches, but communicates rarely with other project members, while B does not submit patches but communicates extensively with other project members. Our results, presented in this article, show that B's activities are more significant for predicting her future committership than are those of A.

It is important to note that this is just the basic framework of how a contributor becomes a committer in ASF projects, and the situations may vary for different projects, or different stages of evolution of different projects. For example, some projects may require developers to enter bugs into an issue tracking database, such as Bugzilla or Jira, while others may require users to submit bug reports to a mailing list. The latter method may increase the interaction between users, developers and committers in those projects, thus affecting the dynamics of earning trust. Additionally, different contributors may have different motivations to join a project [**18**, **39**, **40**], *e.g.,* enjoyment, reputation building, and skill improvement. The commercial backers of some OSS projects may also provide incentives for skilled programmers to contribute in order to grow their project in its early stages, while in the later stages, when the project has matured, developers are often more willing to volunteer in order to gain a signaling benefit to prospective employers [**39**].

3.1.2. Research Questions. In this article we seek to identify effective social activity predictors of developer initiation in OSS projects, with a focus on ASF projects, and to improve upon existing models for predicting future committers. Previous work has focused mainly on technical activity predictors of developer initiation (*e.g.,* based on code commits, patches submitted, or issues resolved). We hypothesise

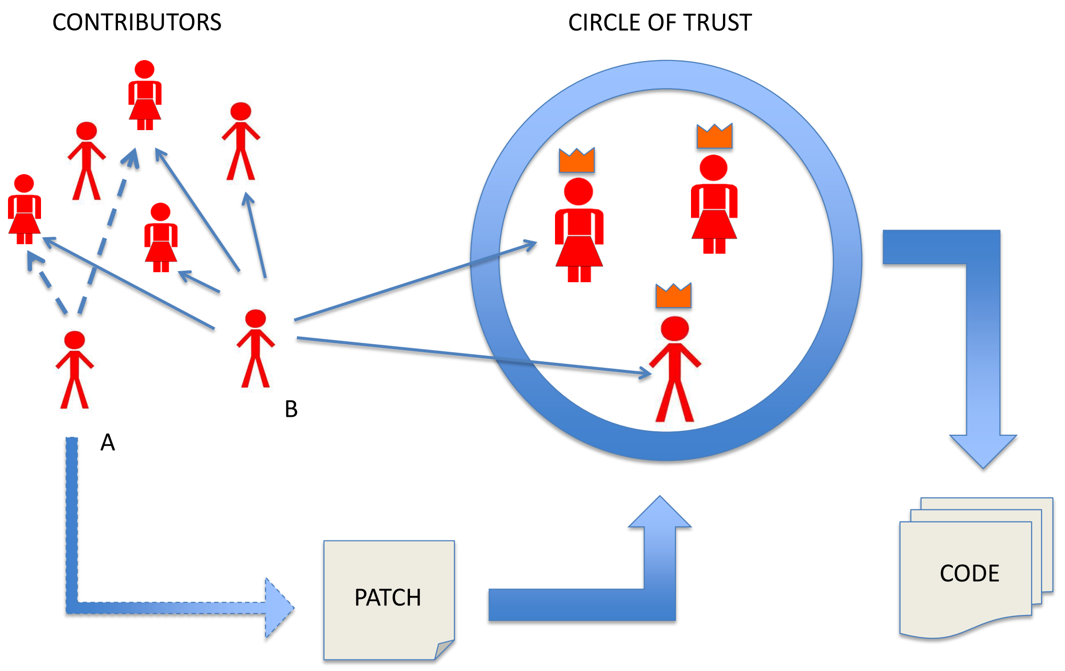


FIGURE 3.1. We consider an ASF project community's circle of trust to be comprised of committers. Developers A and B are candidates for entering the circle. The activities of B are more significant positive predictors of her gaining committer status because she communicates more (thin solid arrows) than A does (thin dashed arrows), even though A contributes patches and B does not.

that social activity yields at least as strong predictors as technical activity, especially in OSS communities such as Apache, which "values the community more than the code"[5].

First, we ask which social metrics are effective predictors that a contributor will become a committer, and how do they interact with the technical measures of patch activities?

Research Question 1: To what extent can developer initiation in OSS projects be modeled as a function of patch activities and social communication? And to what extent solely as a function of social communications?

Early in their tenure as OSS project contributors, people's patterns of technical and social activities are rapidly changing. Contributors usually take some time to familiarize themselves with the code before submitting patch fixes. Additionally, they also might try to assimilate the project culture and available knowledge initially before asking questions of their own. Therefore, predictions of future status may (or

[5]http://community.apache.org/contributors/

may not) be unreliable early in a person's tenure. Here, we seek to predict project participants' likelihood of becoming a committer based on the patterns of their activities early in their tenure, and in doing so understand how early can predictions be done, with reasonable reliability.

Research Question 2: How accurately can developer initiation be predicted from their earliest activities in the project? That is, can we tell if someone will become a committer based on their activities in the first three months? Six months? Or as little as one month?

As with many deadline oriented tasks, approaching the goal is usually associated with the anticipation of it. In that regard, we would expect not much variance in people's behavior just before being initiated a developer, as one tends to ramp up one's activities to reach a goal. The communication activities that follow the initiation time would presumably be less cohesive, and more connected to individual preferences and styles of work. This may be more apparent for people who already participate in other OSS projects. Whereas in the RQs above we contemplate the importance of social communication to achieving committer status, next we ponder at a finer resolution how the social activity of future committers changes as they approach their developer initiation time, and how their communication patterns evolve hence. Specifically, we ask,

Research Question 3: What is the relationship between the amount of one's communication activity and the periods of time just before and just following their initiation as committers? Do they become more social as they approach their initiation time? Do they fall back into more individualized patterns after? Is their behavior related to their experience in other projects?

Finally as projects evolve, determinants of trust are also likely to evolve and change, consequently, measures of trust and predictors of status change may not be static. This is mirrored in other fields, *e.g.,* clandestine operations, where communication is over public channels but action traces are rarely readily observable. While the number of committers in a project typically grows proportional to its size, the number of participants in a project's mailing list (*e.g.,* developers and users) grows exponentially. This increasing gap between potential committers and new position openings in turn change how trust is earned as a project matures.

Research Question 4: Is it easier or more difficult to become a committer later in the project?

3.2. Related Work

In this article we model developer initiation and study social interactions between contributors to six Apache Software Foundation projects. References to related work pertaining to individual steps in our analysis process can be found throughout the text. In this section we discuss three other areas of related work, studies of ASF projects in Section 3.2.1, studies of developer initiation in OSS in Section 3.2.2, and general studies of newcomer incentives and socialization within an organization Section 3.2.3.

3.2.1. Studies of ASF Projects. The Apache Software Foundation has a rich and public history that has driven many research studies. The number and scope of these projects is too large to give full justice here, we mention just a few to illustrate the variety of contexts in which these projects have been studied. Bird *et al.* used Apache software in his work on social network mining [**41**]. Rahman *et al.* used Apache projects to study cross project defect prediction, and bias in software engineering datasets [**42**,**43**]. Jureczko and Madeyski also studied cross project defect prediction using several Apache projects [**44**]. Jureczko and Spinellis used a collection of Apache projects to model defection prediction using Object Oriented metrics [**45**]. Posnett *et al.* used 18 different ASF projects to illustrate the risk of ecological fallacy in empirical software engineering studies [**46**].

3.2.2. Studies of Developer Initiation in OSS. There have been a fair number of studies on the motivations of developers for joining OSS projects and migration in OSS projects. Some projects have clear guidelines on how a new participant can contribute. The structure of this hierarchical process is known in the literature as the “onion model” [**18**,**19**].

Von Krogh *et al.* performed a detailed case study of the Freenet project; they interviewed participants and developers recording their patterns of individual activity and concluded that individuals following these guidelines are highly more likely to become developers [**47**]. Ducheneaut examined a single individual and his process of promotion to a core developer in the Python project [**21**]. Jensen and Scacchi studied role sets and the process of role migration in Mozilla, ASF and Netbeans [**27**].

Sinha *et al.* studied how developers enter the “circle of trust” by identifying key factors that lead to committer status [**26**]. They hypothesized that developers who contribute to the projects’ bug tracking system, have prior experience contributing code to OSS, and who work for the same organization as some member of the core group, are more likely to obtain committer status.

The path to becoming a committer is not necessarily a step-by-step process. Herraiz *et al.* found that apart from gradual progression, there is another common developer joining pattern, *viz.*, the quick initiation of employees of enterprises invested in that OSS project as new developers [**48**]. Shibuya and Tamai performed case studies and confirmed these findings on other OSS projects (GNOME, OpenOffice.org, MySQL) [**49**].

Qureshi and Fang have identified different classes of committers based on socialization patterns using Growth Mixture models. They found that for each class of social behavior, the *"Lead Time"* (*i.e.,* the time it takes to become a developer) is unique and correlates with the amount of social activity of that class [**50**].

Bird *et al.* quantitatively modeled the relationship between time spent with the project and the probability of becoming a committer; their model used patch activities, social network attributes, and the time to first commit from the time of first communication on an email network [**28**]. Using proportional hazard rate modeling they observed that a committer's *tenure* is related to his skill and commitment as measured by his participation in the email network and his contribution of patches prior to first commit. Further, they identified and described a non-monotonic trend in the likelihood of becoming a committer that rises with tenure, peaks, and then declines with project maturity. While their work is similar to ours in some aspects, their approach of predicting the "time" until one becomes a committer is in contrast to our work in that we focus on identifying "who" is more likely to become a committer rather than "when". The hazard analysis techniques used in their work support their approach.

Zhou and Mockus have modeled the status of "Long Term Contributors" based on three dimensions: environment, willingness, and capacity [**51**]. Their work focuses on issue tracking systems and workflows within those systems as sources of information.

Our work differs in that we focus on understanding how soon can we predict developer initiation after initial participation and, additionally, to what degree can social metrics replace technical attributes.

3.2.3. Genereal Studies of Newcomer Incentives and Socialization. The dynamics of social activity patterns around times of precisely defined goal achievements have been studied formally before. In a broader context, the issue is related to reward sensitivity, "an incentive motivational state that facilitates and guides approach behavior to a goal" [**52**], also referred to as goal-directed approach behavior [**53**]. Particularly interesting are the works of Lucas *et al.* [**54**] and Ashton *et al.* [**55**] expounding the importance of reward sensitivity for explaining one's social behavior. In addition, goal-directed approach behavior has been studied in the context of gamification [**56**]. Cheng and Vassileva [**57**] observed "gaming" behavior in an educational online community, *i.e.,* incentivizing user contributions with status enhancements (analogous to incentivizing developers with committership prospects in our context) motivates users to adjust their behavior to maximize their chances of receiving the reward, even inappropriately by adding low-quality resources. Farzan *et al.* [**58**] found that some incented users stop contributing after reaching a specific status level in a social networking website for employees at IBM. Anderson *et al.* [**59**] studied badges, or online distinctions achieved by users as reward for activity, and their role in steering online users behavior in Stack Overflow. They found that "activity on the targeted actions increases substantially before users achieve the badge, and then almost immediately returns to near-baseline levels. Most of the other site actions are not adversely affected—the rates of these actions remain relatively stable over time." Similar findings have been reported by Grant and Betts [**60**].

While our results are specifically about achieving committer status within open source projects, it fits within a more general framework of understanding how newcomers are integrated into their work environment. Begel and Simon studied newcomers to software development within Microsoft [**61**]. Their focus was on understanding how science pedagogy prepares students for the workforce. One finding of interest to this study is that they found that newcomers spend significant portions of their time communicating with peers. The process of transitioning from outsider to insider is called *Organizational Socialization*, or *Onboarding*, in the psychology literature [**62**,**63**]. Baur *et al.* study the orgnaizational socialization of 70 individuals using path modeling to ascertain how the effects of role clarity, self-efficacy, and social acceptance, mediate the effects of organizational socialization tactics and newcomer information seeking [**64**]. A finding of interest here is that social acceptance has a positive mediating affect on newcomer information seeking with respect to performance, organizational commitment and intentions to remain.

3.3. Data Gathering

The Apache Software Foundation is an umbrella for hundreds of different OSS projects. We sought to analyze a diverse sample of them, with respect to both project size and activity. We selected six ASF

TABLE 3.1. The ASF projects selected in this study show diversity both in size and in relative activity. #Users refers to the number of individuals in the email social network. #Committers refers to the number of distinct committers to each project's source code repository.

Project	#Users	#Committers	#Mails	#Patches	Start	End
Ant	1416	44	17300	1482	2000-01	2012-03
Axis2_c	600	24	11152	754	2004-01	2012-03
Log4j	539	18	3811	166	2000-12	2012-03
Lucene	2155	41	43922	5576	2001-09	2012-02
Pluto	266	24	3017	259	2003-10	2011-09
Solr	840	19	14411	4090	2006-01	2010-04

projects, each with an acceptable minimum number of contributors, to make our modeling results statistically acceptable. The six projects together with summary statistics are presented in Table 3.1.

For each project we mined data from three sources. The first is the *developer mailing lists*, which contain traces of developers *social activities*, from which we reconstruct email social networks. We deemed the mailing lists provide an unbiased set of communications, sufficient for our purposes of capturing neccessary data for our statistical predictive models[6]. The second is the *issue tracking systems*, which contain information about patch submissions, indicative of one's *technical activities* pre-committership. And third, we mined the *source code repositories* to extract information about committer status. All sources were mined from the earliest date the data was available, until the date of mining (March 2012). The start and end dates reported in Table 3.1 represent the intersection of available data from all three sources, for each project.

In this section we describe the specific process we followed to extract data from each of the three sources. An overview is depicted in Figure 3.2. All data (processed and raw) described here along with all the scripts used to process them are available in an online appendix, at `http://csiflabs.cs.ucdavis.edu/~ghareh/supplementary/oss/`.

3.3.1. Unmasking Aliases. OSS contributors often use different aliases (combinations of names and email addresses such as `<JohnSmith,smith@gmail.com>`, `<Smith,John@smith.com>`, `<JohnS.,J.smith@ucdavis.edu>`) in different repositories they participate in (*e.g.,* source code repositories, issue trackers, mail archives), or even in the same repository but at different times (*e.g.,* an ASF committer with a personal, say `gmail.com`, email address might also push changes from an account configured to use her `apache.org` email address). Since such aliases represent a single physical entity (*i.e.,* a person), they must be merged if one is to accurately capture a contributor's total activity within the project.

[6]Issue trackers also capture communication between committers and developers. We did not use those because the mailing lists contained a large enough communication sample which was not obviously biased in any way

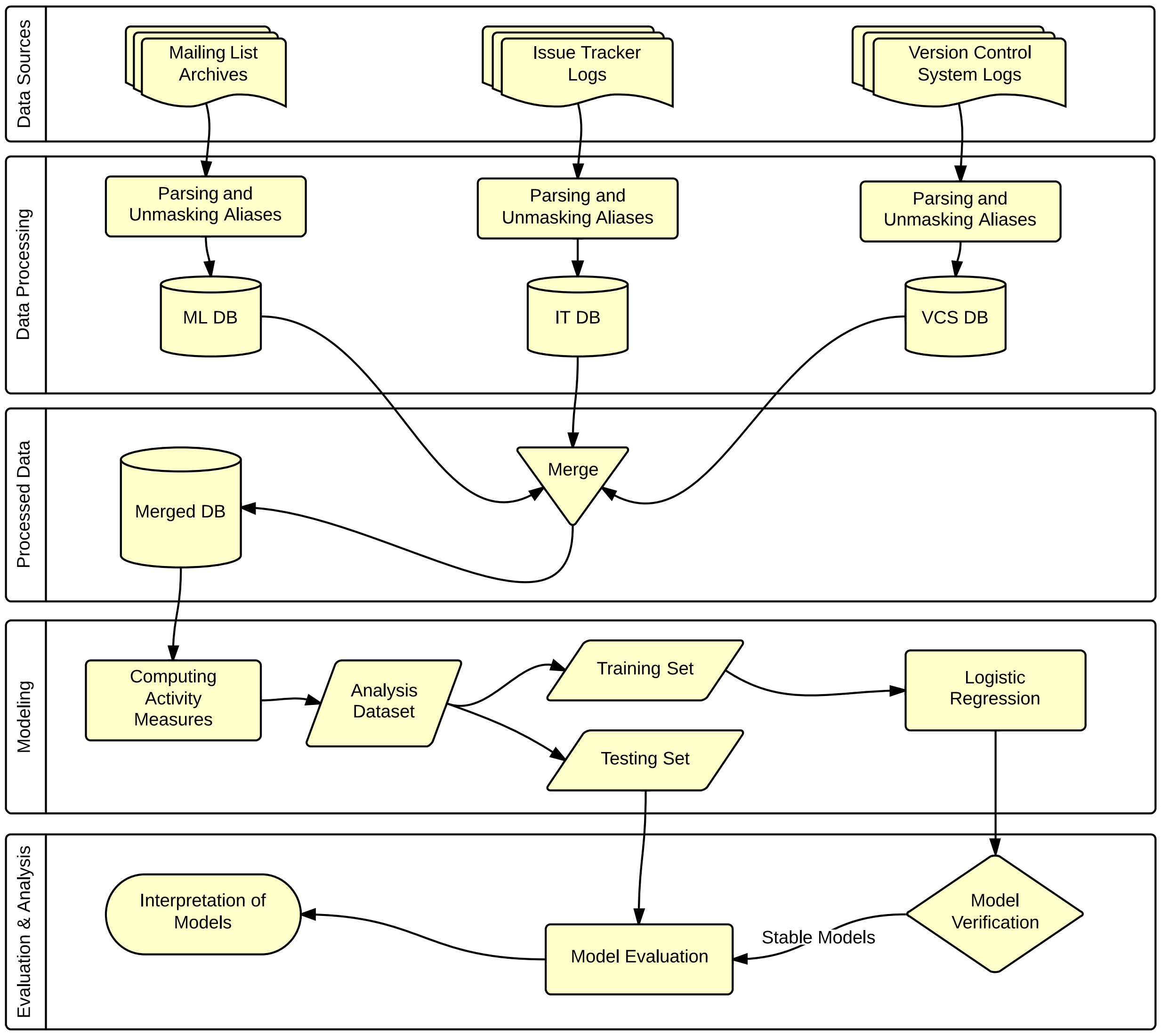


FIGURE 3.2. The process diagram of data gathering, processing, modeling, and finally evaluation as described in the paper.

Unmasking aliases (or identity merging) is a well-recognized problem in the literature (*e.g.,* [**41**, **65**, **66**]), thus far without any perfect solutions [**67**]. In this article we employ an extended version of the technique developed by Bird *et al.* [**41**], based on heuristics and string similarity measures. The heuristics include *guessing* a person's likely email address prefixes based on their first and last names (*e.g.,* John Smith might use prefixes such as `john`, `jsmith`, `johns`, or `john.smith`), then using this information to aid matching the different aliases. Our extension consisted in refining the heuristics (*e.g.,* we have observed in our dataset that certain email addresses such as `jira@apache.org` may be used by different contributors;

therefore, such addresses should not be used for matching) and automating the procedure further, to reduce the need for human interaction.

Succinctly, the process consisted of the following steps. First, we filtered names and removed all suffixes, prefixes, and generic names, such as Dr., Mr., Jr., or Admin. Then, for each pair of aliases we calculated a similarity score (on a unit scale) based on the Levenshtein (edit) distance between the different alias parts (name-to-name, or name-to-email-prefix), as described in the original work by Bird *et al.* [**41**]. Next, we merged perfect matches (having score 1) automatically, and presented less than perfect matches that achieved a score of at least 0.93 (threshold determined empirically on other Apache data to offer a good tradeoff between false positives and false negatives) to one of the authors to disambiguate. Finally, the results were reviewed by a different author and few incorrect matches were corrected.

The process was repeated for each of the three mined data sources (source code repositories, issue trackers, mail archives), with more manual intervention of the authors to merge aliases *between* the three.

3.3.2. Constructing Email Social Networks. It is a common policy for OSS projects to channel as much communication as possible through project mailing lists so that all participants can benefit from the exchange of ideas and information [**41**]. While other venues of communication in OSS exist (ranging from discussions around issues recorded in bug tracking systems, to IRC channels, to private email or even offline communication between developers), developer lists are known to be the hub of developer communication in OSS [**68**], *i.e.,* the place where most communication happens and where the team meets to discuss issues, code changes, additions, etc. Therefore, we view developer lists (such as `dev@ant.apache.org` mined in this article) as representative of the different developer communication media available in OSS. In addition, developer lists are very suitable for studying developer initiation in ASF since all proposals and voting for granting developers committership are recorded therein.

Any message sent to a mailing list will be broadcast to all subscribed participants. Still, point-to-point exchanges (links) between different list subscribers can be inferred, using the reply information [**41**, **69**, **70**]. When person B replies to a message originally broadcast by person A to the list, there is a high chance that B primarily intended to communicate directly to A as opposed to again broadcast to the entire list [**41**]. Such links are in fact two-way social links, in that the communication occurs both ways (A communicated with all list subscribers hence implicitly also to B; B communicated explicitly to A since she replied directly to A).

The networks resulting from parsing the emails in each project's mail archives, reconstructing the discussion threads and extracting the in-reply-to links are known as Email Social Networks (ESNs) [**41**]. ESNs are indicative of a project's social structure and can be used to assess a contributor's *social* activity. We created ESNs for each of the six projects by removing self-loops, *i.e.,* replies to one's own message, and allowing multiple edges between people (*e.g.,* as inferred from communication at different times). We use the ESNs to compute social activity measures for each developer, such as the total number of messages sent within a time unit, the number of neighbors, or the number of neighbors having committer status, as detailed in the Modeling section below.

3.3.3. Mining Patch Submissions. Patches are small pieces of code intended to fix bugs or otherwise incorporate small changes into the code base. While patch submission is not limited to non-committers, the term "patches" is typically used to denote code contributions by developers who have not yet been granted commit rights to a project's source code repository. Patch submission does not imply patch acceptance, with committers reviewing and applying patches as they see fit. Patch submission does however signal one's interest to contribute to a project, and is a typical *technical* means for a developer to build reputation within a community before reaching committer status.

There are multiple methods and formats in which one can submit a patch to an OSS project. The commonly employed format for a patch submission is a "diff" file containing the proposed changes. Patches are submitted for review either as attachments to issues opened in the project's issue tracking system (*e.g.,* Jira, Bugzilla), or as attachments to messages posted on the developer mailing list. Furthermore, since patches are in fact source code fragments, they can be embedded directly in the text of a comment posted to an open issue or, similarly, in the text of a message sent to the developer list. For ASF projects, all these methods for patch submissions are common.

File names of patches submitted as attachments typically end with a `.diff` or `.patch` extension. This convention is not always respected, as we also found patches submitted under different names (*e.g.,* `patch.txt`), or multiple patch files combined in an archive. To capture this variance, we developed a set of heuristics based on regular expression pattern matching, that included inspecting the contents of archived files. Similarly, we used regular expression pattern matching queries to extract inline patches from emails, or comments or descriptions to issues reported in Jira and Bugzilla. A pseudocode representation of the

```
bool isMessageAPatch(message m) {
    messageText = getMessageText(m);
    diffExpressions = ["+++", "---"]
    if (contains(messageText, diffExpressions)==T)
      return(TRUE);
    attachments = getMessageAttachments(m);
    for (attachment in attachments)
      if (isAttachmentAPatch(attachment) == TRUE)
        return(TRUE);
    return(FALSE);
}

bool isAttachmentAPatch(attachment A) {
  patchExpressions = [".patch", ".diff", "patch."]
  if (contains(fileName(A), patchExpressions) == T)
    return(TRUE);
  else
    if (isArchive(A) == TRUE) {
      files = extract(A);
      for (file in files)
        if (isAttachmentAPatch(file) == TRUE)
          return(TRUE);
    }
  return(FALSE);
}
```

ALGORITHM 1. The pseudocode for mining patches from message texts and attachments

TABLE 3.2. Different projects have different practices for patch submission, making extraction challenging.

	Maling Lists		Bugzilla		Jira	
	Inline	Attachments	Inline	Attachments	Inline	Attachments
Ant	413	976	0	93	0	0
Axis2_c	63	200	0	0	27	464
Log4j	87	57	0	19	0	3
Lucene	141	107	0	0	88	5240
Pluto	15	26	0	0	8	210
Solr	30	4	0	0	48	4008

processes used to mine patches is given in Algorithm 1. The distribution of patches mined in each project from the different sources is presented in Table 3.2.

3.3.4. Mining Source Code Repositories. A source code repository and a version control system, *e.g.*, Git, SVN, and CVS, facilitate collaboration among committers by maintaining a history of changes and an

TABLE 3.3. The total number of committers to each project, out of which those that do not appear in the ESN or have less than three months of social activity data available prior to them becoming committer (the *filtered* column). Remaining committers are what we consider *initiated developers*.

Project	#Committers	#Filtered Committers	#Remaining Committers
Ant	44	13	31
Axis2_c	24	3	21
Log4j	18	8	10
Lucene	41	9	32
Pluto	24	10	14
Solr	19	5	14

associated log entry for each change. These systems can provide various information about a project's size (*e.g.,* a list of all the files), team size (*e.g.,* a list of committers), or a detailed record of all changes induced by any committer. Currently, all six ASF projects considered here use Git as their version control system, but some initially used either CVS or SVN and later migrated to Git. The current Git logs incorporate the history of changes to these projects prior to their migration to Git.

Among others, this information allows us to distinguish between *developers* (contributors that are part of the ESN and submit patches through any of the methods described above) and *committers* (contributors who push changes directly to the project's source code repository as evident from the version control logs), at any time in the history of a project. Consequently, we consider the date of one's first recorded commit in the project's version control logs as their date of becoming a committer. While this is an approximation, since a contributor may have been granted committership before her first actual commit, both previous work [**28**] as well as manual inspection of a random sample of developers in our dataset suggest this to be an accurate representation of one's initiation date.

Table 3.3 lists the number of distinct committers (after unmasking aliases) for each project, as well as the number of distinct committers used in the statistical modeling (described next), *i.e.,* those present in the project's ESN and for which at least three months of social activity data is available prior to becoming committers.

3.3.5. Computing Social and Technical Activity Metrics. Gathering the data from the three sources described above resulted in a rich longitudinal dataset of social and technical activities in each project. Figure 3.3 provides an illustration of this dataset, with the unfolding of events in one project involving three contributors A, B and C. Assuming the project started at time t_1, we observe that B commits to the

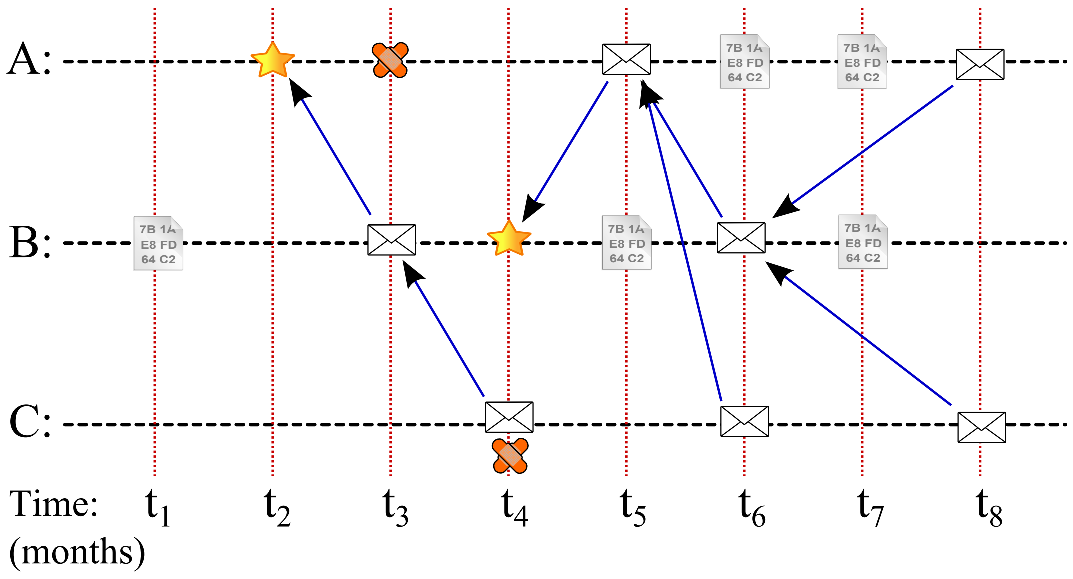


FIGURE 3.3. Illustration of the interplay between social and technical activities in our dataset. The metrics for this sample are given in Table 3.4. A star denotes starting a new thread, the envelope symbolizes responding to a thread/message, the patch icon denotes submitting a patch and the code icon symbolizes committing code to the repository. An arrow from A to B denotes that B's message was sent in response to A's. The horizontal axis denotes time.

TABLE 3.4. Social and technical activity measures for sample dataset in Figure 3.3 assuming $k = 3$.

ID	Initiated	Num. Patches	Num. Messages	Num. Threads	Neighbors	Neighbor Committers	Project Age
A	TRUE	1	5	1	2	1	90
B	FALSE	0	6	1	2	1	90
C	FALSE	1	3	0	2	2	120

repository during the same month (hence B starts off as a committer). At time t_2, A starts a new discussion thread on the mailing list. At time t_3, B replies to the thread previously started by A. Shortly after, A attaches a patch to an issue reported by someone else in the issue tracker (not shown). At time t_4, C joins the discussion thread started by A at t_2, by replying to the email sent by B at t_3 and attaching a patch to this reply email. We skip ahead to time t_6, when A's first commit is recorded in the version control logs, hence t_6 becomes A's initiation date as committer. In contrast, C continues being active on the mailing list until t_8, the end of time displayed, but without ever being initiated as committer.

Note that we will build different prediction models for one's likelihood of becoming a committer (time-independent binary outcome variable) using data about their communication and patch submission activities collected during their first k months of activity in the project (we experiment with $k = 1$, $k = 3$, or $k = 6$, recall RQ2). This implies that a developer's communication or patch submission activities must have preceded their initiation date as committer by at least k months, otherwise they would be excluded from the sample. For example, in Figure 3.3 B would be excluded at any of $k = 1$, $k = 3$, or $k = 6$, while A would be excluded at $k = 6$ because her activity started at t_2 and her initiation date is t_6.

On the dataset resulting from the above pre-processing, we compute the following measures of social and technical activity for each developer (for the example in Figure 3.3, the measures are given in Table 3.4):

- *Initiated*: A binary variable indicating whether this developer ever committed directly to the repository *after* her first message in the mailing list.
- *Number of Patches*: The total number of patches submitted during the first k months of activity in the project (months are approximated as 30 days).
- *Number of Messages*: The number of edges connected to a node in the ESN, *i.e.,* a node's degree. This is not just the number of messages one sends, but rather the number of *replies* one **sends** plus the number of replies one **receives**.
- *Number of Threads*: The number of threads started. A thread is a message that is *not* sent in response to any other messages.
- *Neighbors*: The number of unique nodes that a node is connected to in the ESN. This is different from *Number of Messages* in that a node can connect to other nodes through multiple edges.
- *Neighbor Committers*: The number of unique nodes with commiter status that a given node is connected to in the ESN.
- *Project age*: The number of days from the start of the mailing list to the first appearance of this node, *i.e.,* the first message received by or replied by that person in the ESN. In our example both A and B's *Project age* are 90 days because as explained previously, only response messages are counted towards the ESN and for A it is the first response it gets that counts as its first link in the ESN.

3.4. Modeling

In this section we describe our methodology, including logistic regression and the analysis of socialization dynamics around the time of becomming a contributor.

3.4.1. Modeling Committer Initiation. We use logistic regression, a generalized linear modeling technique designed to model probabilities for dichotomous outcomes, to model whether or not a developer will reach committership based on several social explanatory variables.

The logit $log(\frac{1}{1-p})$ models unbounded response as a probability using *maximum likelihood* estimation which, given a distribution, finds the values of the parameters that give the observed data the greatest probability. That is, maximum likelihood is used to estimate the following general model which yields an estimated probability $\hat{p_i}$ that the true value of the response is 1 [**71**].

$$log(\frac{1}{1-p}) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_p x_p + \epsilon$$

For each predictor the *z-test statistic* is computed. This statistics divides the estimated value of the parameter by its standard error and is used to assess the significance of the variable. This statistic is a measure of the likelihood that the actual value of the parameter is not zero [**71**].

Previous work in this area has used survival modeling to model the trajectory over time of developer initiation [**28**]. In this work we are focused on early detection of developer initiation which limits the amount of data available to model the trajectory. Moreover, since we are interested in the dichotomous outcome of whether a participant becomes a developer, logistic regression is a more appropriate choice.

The dataset used for our studies, contains the features and metrics described above for the first k months, of each individual's activity ($k = 3$ unless explicitly stated). We attempt to predict the *"Initiated"* outcome. We generated multiple models for each of the research questions. The variables used for each of the models are described in the results.

For our models we are primarily interested in the direction of the effect of each predictor. We want to control as much as possible for other sources of variation that might be incorrectly attributed to the variables of interest.

The age of the project or "Project age", *i.e.,* when a person joins the project, is added to all models as a control variable. For all numeric variables, the $\log$ of that variable plus 0.5 was used to stabilize variance and reduce heteroscedasticity [**71**]. Since all untransformed values in our data are skewed, the increase in the value of a variable by half a unit does not have the same effect at high values as it does in low values, *e.g.,* the number of patches changing from 1 to 1.5 is much more meaningful than changing from 100 to 100.5.

For transformed variables, comparison with the non-transformed variable shows that this transformation yields a better fit using Vuong's non-nested test [**72**].

Considering we are trying to predict who *is going to* become a committer, sample data for developers who were initiated in fewer than k months were removed from the dataset (Table 3.3).

For each learned model, we evaluate its validity based on two criteria. The first is project independence, *i.e.,* we want our model to hold across projects. One way to address this issue is to merge all projects' datasets into one dataset. This raises several concerns such as scaling of variables *i.e.,* 30 messages may considered much in one project while it is a small value in another. At the same time multiple project dependent parameters that we cannot measure such as project "culture", or simply we are not aware of exist. The other solution is to look at the stability of each model coefficient's statistical significance, as determined by the coefficient's *p-value*. We choose the latter because it does not pose the aforementioned threats.

Commonly, a p-value of less than 0.05 is an indicator of significant results. If a predictor's coefficient is statistically significant across all projects, then it is more likely to be a project independent factor. On the other hand, if a coefficient is only statistically significant on few of the projects, it is most likely dependent on some project parameter, such as "culture", and "maturity".

Excessive multicollinearity is a concern in regression models and it can occur when predictors are highly correlated. To check for this we use the *Variance Inflation Factor (VIF)*. A common rule of thumb is that for any variable x in a model, $VIF(x) \geqslant 5$ indicates high collinearity. In all of our models in this article, VIF of all variables remained well below 2 except for some models with highly correlated variables. These models were discarded as it will be explained later in the article. We then move towards evaluating each model's performance *i.e.,* predictive power.

Some of the basic definitions of a binary classifier's performance are True Positive (TP), True Negative (TN), False Positive (FP) and False Negative (FN). True Positive/Negative is the number of positive/negative samples that the classifier guessed correctly. False Positive/Negative is the number of negative samples that the classifier guessed wrong. TP rate is defined as TP over number of all actual positive samples in the testing set and FP rate is defined as FP over the number of all actual positive samples. The Receiver Operating Characteristic or the ROC curve illustrates the performance of a binary classifier in a *FP rate-TP rate*space, while varying the cutoff threshold. A random predictor would be a line with the slope of 1 and the area of 0.5 while a perfect predictor will have an area of 1 because it will always have a TP rate of 1, and an FP rate of 0.

To evaluate a model's predictive power, we use the *Area Under the Receiver Operating Characteristic (AUROC)* measure [**71**]. For each model, we measure its AUROC and the closer this value is to 1, the better the predictor will be. In cases where we aggregate performance of many models, we simply measure the mean of the models' AUROC.

Overfitting is a concern with any statistical model so to help alleviate any concern and to yield a stable estimate of the predictive power of our models we employ resampling methods. We define training and testing sets using $2/3$ holdout for our training sets. To maintain a similar distribution of committers vs. developers in the training and testing sets we employ stratified sampling. Each of the test and training sets will then have roughly the same ratio of committers to developers.

This ensures that the resulting model is not extremely biased in that the training set would contain almost all, or none of the developers. The first case would cause a testing set with no positive samples, and the latter would results in a zero model due to the lack of positive samples in the training set. We resample 250 times and average the AUROC over all these models to indicate the overall predictive power of a model.

3.4.2. Dynamics of Social Activities Before and After Becoming a Developer. Here we look at the dynamics of the amount of socialization, or social activities, *i.e.,* emails, of developers around the time of them becoming a committer. We start by counting the number of messages sent to or received by a person in the 6 months prior to their developer initiation and the 6 months immediately after the event. We filter out those developers who are not active in at least 6 months before and after, to avoid biased results (the number of remaining developers is given in the results). Then, we can observe the overall behavior of the developer population by comparing the number of messages before and after becoming a developer. By aggregating the data in buckets we can statistically assess how socialization changes from the period before to the period after initiation. In addition to the statistics we also conduct a case study on a handful of developers to offer email content-based evidentiary support for our findings.

3.5. Results and Discussion

3.5.1. Research Question 1. We evaluate here the stability and predictive power of models using patches, length of time with the project, and a number of social measures. We motivate this question with Figure 3.4. This figure shows that, although most of the time there is a meaningful difference between committers and developers who submit patches, still there are many developers who behave like

TABLE 3.5. Patch submission is a significant predictor when no social variables are included in the model. This basic logistic regression model only uses "number of patches" in 3 months and includes "project age" as a control variable. For all variables, the $\log$ of that variable plus 0.5 was used in the modeling. The values in the first column are the model coefficients and the highlighted coefficients are statistically significant ($p < 0.05$).

Ant	Estimate	Std. Error	z value	Pr(> |z|)
(Intercept)	-1.73	1.12	-1.55	0.12
Project age	-0.33	0.18	-1.87	0.06
Number of patches	**1.06**	0.18	5.82	0
Axis2_c	Estimate	Std. Error	z value	Pr(> |z|)
(Intercept)	-0.91	1.01	-0.9	0.37
Project age	**-0.39**	0.16	-2.46	0.01
Number of patches	**0.93**	0.21	4.53	0
Log4j	Estimate	Std. Error	z value	Pr(> |z|)
(Intercept)	-3.53	1.98	-1.78	0.07
Project age	-0.11	0.29	-0.38	0.7
Number of patches	0.36	0.83	0.43	0.67
Lucene	Estimate	Std. Error	z value	Pr(> |z|)
(Intercept)	1.08	0.84	1.28	0.2
Project age	**-0.7**	0.12	-5.71	0
Number of patches	**1.16**	0.18	6.42	0
Pluto	Estimate	Std. Error	z value	Pr(> |z|)
(Intercept)	-0.88	0.84	-1.05	0.3
Project age	**-0.34**	0.15	-2.3	0.02
Number of patches	**1.11**	0.32	3.45	0
Solr	Estimate	Std. Error	z value	Pr(> |z|)
(Intercept)	0.44	1.19	0.37	0.71
Project age	**-0.72**	0.19	-3.69	0
Number of patches	**0.75**	0.29	2.58	0.01

committers in terms of patch submission. Consequently, it is likely that using patches alone may not yield the best prediction model.

This simple model, using "Number of patches" and no social network measures shows that patch submission is often a statistically significant predictor (Table 3.5). Based on what was described in the methodology section the prediction model's formula in this case is:

$$Initiated \sim \log(project\ age + 0.5) + \log(number\ of\ patches + 0.5)$$

In Table 3.5 (as well as all other tables with same format hereafter) we have shown the results of logistic regression on each projects data separately. Each row below each project represents that specific predictor. The "Estimate" column represents the estimated coefficient for that specific predictor. "Std. Error" is the

TABLE 3.6. The second logistic regression model, adding "number of messages" to the previous model: $\textit{Initiated} \sim \log(\textit{project age}+0.5)+\log(\textit{number of patches}+0.5)+\log(\textit{number of messages}+0.5)$. It is seen that "number of patches" slightly loses its significance.

Ant	Estimate	Std. Error	z value	Pr(> |z|)
(Intercept)	**-4.32**	1.32	-3.28	0
Project age	-0.2	0.19	-1.08	0.28
Number of patches	**0.61**	0.2	3.06	0
Number of messages	**1.07**	0.2	5.35	0
Axis2_c	Estimate	Std. Error	z value	Pr(> |z|)
(Intercept)	**-2.85**	1.33	-2.15	0.03
Project age	-0.3	0.17	-1.81	0.07
Number of patches	**0.53**	0.25	2.11	0.03
Number of messages	**0.57**	0.22	2.62	0.01
Log4j	Estimate	Std. Error	z value	Pr(> |z|)
(Intercept)	**-7.28**	2.51	-2.9	0
Project age	-0.13	0.3	-0.42	0.67
Number of patches	-0.8	0.98	-0.82	0.41
Number of messages	**1.89**	0.44	4.26	0
Lucene	Estimate	Std. Error	z value	Pr(> |z|)
(Intercept)	0.26	0.89	0.29	0.77
Project age	**-0.8**	0.13	-6.04	0
Number of patches	**0.69**	0.22	3.14	0
Number of messages	**0.74**	0.2	3.75	0
Pluto	Estimate	Std. Error	z value	Pr(> |z|)
(Intercept)	-1.44	1	-1.44	0.15
Project age	**-0.37**	0.15	-2.46	0.01
Number of patches	0.81	0.42	1.95	0.05
Number of messages	0.42	0.4	1.04	0.3
Solr	Estimate	Std. Error	z value	Pr(> |z|)
(Intercept)	**-2.86**	1.45	-1.97	0.05
Project age	**-0.67**	0.2	-3.39	0
Number of patches	-0.15	0.35	-0.44	0.66
Number of messages	**1.18**	0.31	3.83	0

standard error of the coefficient and "z value" is the t-statistic of the coefficient. "$Pr(> |z|)$" shows the p-value of the coefficient for the predictor.

However, adding "number of messages" (as an indicator of social collaboration) to this model results in patches slightly losing their significance (Table 3.6). We used a Chi-Squared goodness of fit statistic to verify that the additional predictor explained a statistically significant amount of the deviance in the model, *viz.*, is the addition of the new variable justified. For all projects projects except Pluto adding "number of messages" was significant with a Chi-Squared test p-value of < 0.01. A Spearman correlation test between

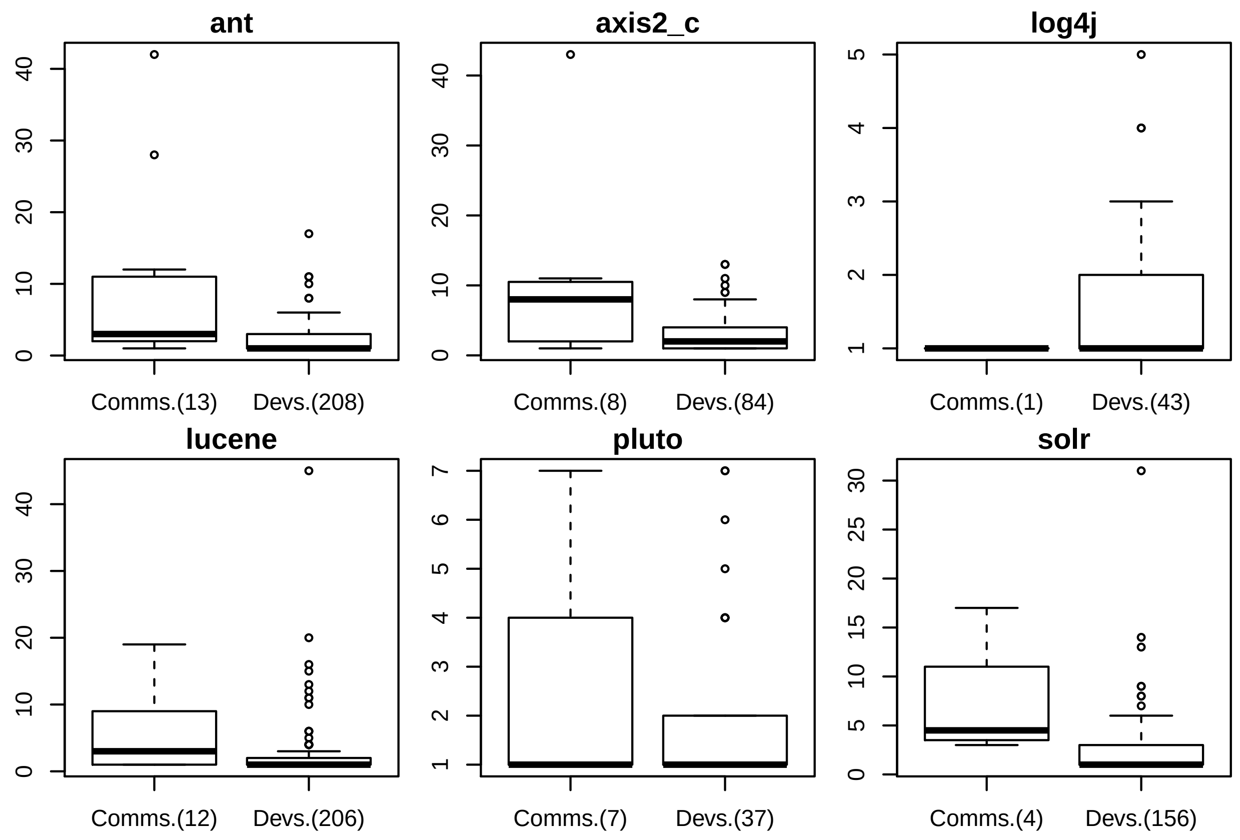


FIGURE 3.4. Distribution of number of patch submissions by committers and developers in each project. Only individuals with at least one patch submission are plotted, since adding those with no submission would highly skew the plots towards zero. The numbers in the parentheses show each group's population (after filtering non-patch-submitters). A Wilcox signed rank test across each project yields p-values in order: 0, 0.01, 0.44, 0, 0.72, 0, indicating that in Log4j and Pluto, patch submission is not statistically different for those participants who become committers and those who don't when measured in the first three months of participation.

"number of messages" and "number of patches" does not show significant correlation between them, mostly below 0.3 over all projects, in concordance with our VIF values of less than 2.

The predictive power of these two models and the additional models with only "Number of messages" and the combination "Number of messages + Threads" is shown in Figure 3.5. We see that not only adding number of messages dramatically improves the predictive power, but removing the patches variable from the model does not lower the predictive power of the model. A Kruskal-Wallace test followed by a post-hoc pairwise Wilcoxon test for each project reveals that in all projects except Pluto and Lucene either the models with messages or messages and threads have the highest mean AUROC, and this difference is statistically significant. In Lucene, the best model uses both patches and messages. In Pluto, although the patches model

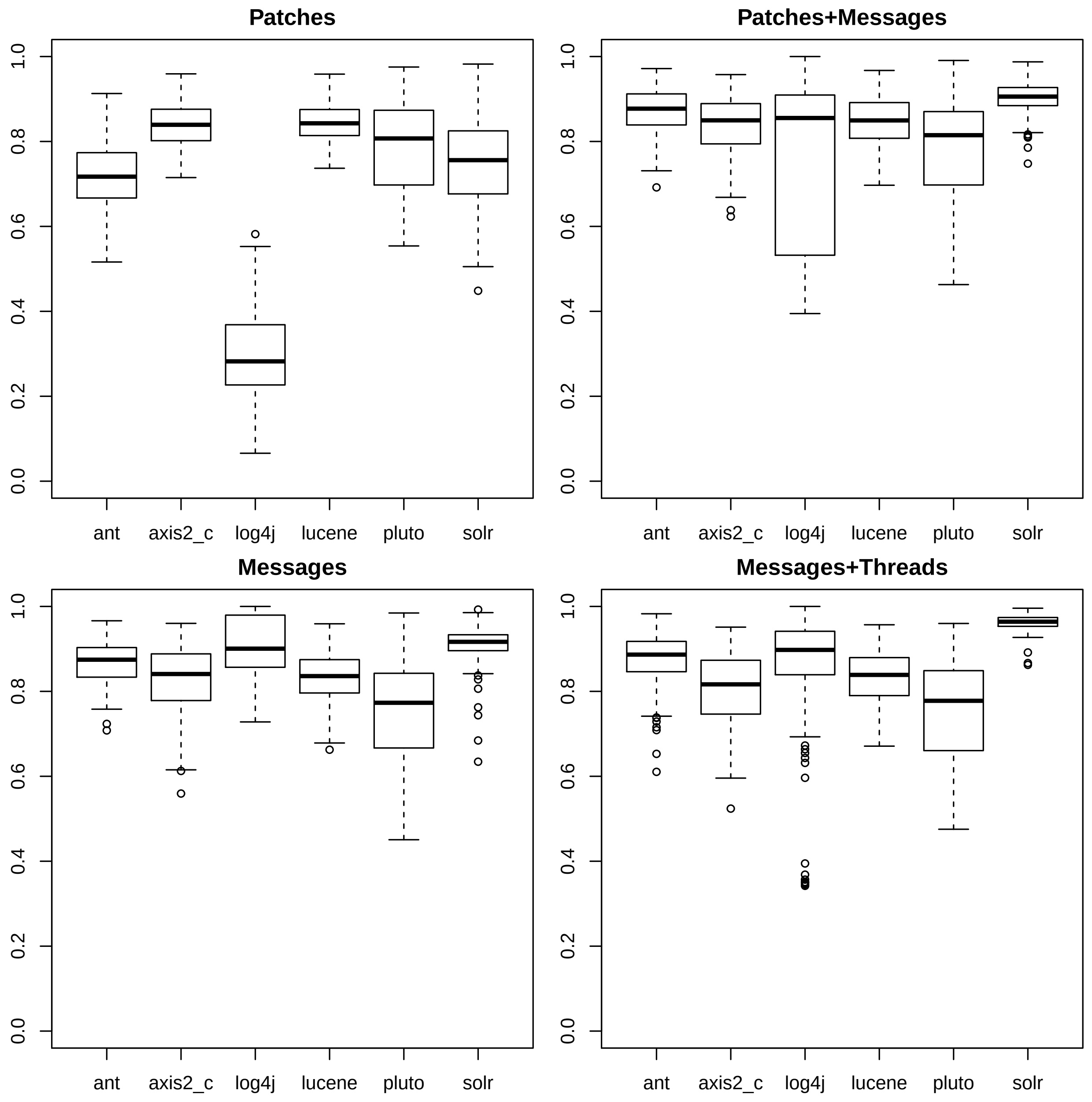


FIGURE 3.5. Social measures outperform patch submission in a predictive setting. AUROC of 4 models, on 250 iterations of modeling using stratified data.

has the highest AUROC, there is no statistical difference between the models. Using patch information alone is not a bad predictor, but it is evident that using social network metrics yields more accurate predictions.

Furthermore, we attempt to improve this simple model by adding additional features from the ESN. Since we have observed that sending and receiving messages is an important indicator of whether someone will become a developer or not, naturally we ask whether it is the number of messages that is important or

TABLE 3.7. Spearman's Correlation between different variables in all projects. Correlation values higher than 0.5 are highlighted.

	Ant	Axis2_c	Log4j	Lucene	Pluto	Solr
messages vs. comm. neighbors	**0.62**	**0.52**	0.49	**0.55**	**0.53**	**0.65**
neighbors vs. comm. neighbors	**0.69**	**0.61**	**0.60**	**0.72**	**0.72**	**0.80**
messages vs. neighbors	**0.82**	**0.76**	**0.70**	**0.67**	**0.66**	**0.75**
threads vs. comm. neighbors	0.21	0.38	0.12	0.38	0.25	**0.56**
threads vs. neighbors	0.31	0.54	0.14	0.37	0.25	0.48
threads vs. messages	0.31	**0.74**	0.17	**0.61**	**0.53**	**0.64**

TABLE 3.8. High multicollinearity limits the effectiveness of additional social variables. None of the added social network measures are stable across projects. Number of threads is significant in two projects, ant, and solr.

	Ant	Axis2_c	Log4j	Lucene	Pluto	Solr
(Intercept)	**-4.13**	**-4.24**	**-6.28**	-0.27	**-2.49**	-2.78
Number of messages	**1.2**	**0.9**	**1.67**	**1**	0.58	**1.01**
Number of neighbor devs	0.08	-0.26	0.25	0.05	0.72	0.27
Project age	-0.29	-0.21	-0.17	**-0.83**	**-0.33**	**-0.65**
(Intercept)	**-5.64**	**-3.78**	**-6.87**	-0.39	**-2.04**	**-5.33**
Number of messages	**1.04**	**0.82**	**1.51**	**1.14**	**0.69**	**2.74**
Number of threads	**0.69**	-0.01	0.6	-0.15	0.28	**-1.26**
Project age	-0.17	-0.26	-0.12	**-0.82**	**-0.43**	**-0.77**
(Intercept)	**-5.7**	**-4.71**	**-16.87**	-1.66	**-2.66**	**-5.64**
Number of messages	**1.01**	0.21	-2.09	0.25	0.23	**2.21**
Number of threads	**0.69**	0.12	**1.07**	-0.03	0.34	**-1.24**
Number of neighbors	0.05	1.12	**7.3**	**1.58**	0.89	1.08
Project age	-0.16	-0.15	0.9	**-0.67**	**-0.34**	**-0.74**
(Intercept)	**-5.63**	**-4.28**	**-7.06**	-0.39	**-2.26**	**-5.56**
Number of messages	**1.03**	**0.94**	1.15	**1.12**	0.34	**2.6**
Number of threads	**0.69**	-0.04	0.68	-0.14	0.35	**-1.27**
Number of neighbor devs	0.02	-0.27	1.02	0.03	0.85	0.37
Project age	-0.17	-0.21	-0.11	**-0.82**	**-0.39**	**-0.72**

the number of distinct individuals one keeps in contact with? More precisely, are these contacts the same, or is communicating with developers more important than communicating with other participants? Also we want to see whether starting threads and discussions in contrast to replying and being replied to, is also an important factor in gaining the trust of the community.

These variables are quite highly correlated and we expect that this will impact model performance (The Spearman correlation between these variables can be seen in Table 3.7). We added the number of started threads, neighbors and neighboring developers to our existing models. While some predictors are statistically significant in some models as can be seen in Table 3.8, most are hampered by the high variance inflation factor owing to the high correlation between "Number of Messages", "Number of neighbors", and "Number of developers" (Table 3.7). "Number of threads", however, was significant in two projects, Ant, and Solr, and had a sufficiently low variance inflation factor that inclusion of the variable improved prediction results. We discuss this further in the next subsection.

Result 1: *Developer initiation can be modeled using social activity alone, performing no worse than models which also incorporate patch submission. The basic model of social activity only uses "Number of Messages", however adding "Number of Threads" improved prediction results in 2 of the projects, hinting this might be a matter of "project culture".*

3.5.2. Research Question 2. In the previous section we used information on the first three months of individuals' activity to model their likelihood of obtaining committer status. But how early can such models provide useful predictions? Is one month of information sufficient, or should we increase to 6 months or more to yield better prediction models?

To evaluate the sufficient-time-for-prediction hypothesis, we use the simple model discussed in the previous section (only using "Number of Messages" as a predictor) with information on the first $n = 1, 2, ..., 6$ months of each participant's activity in the OSS ESN.

A limitation in evaluating models for long time periods is that participants who become developers in a shorter period must be discarded from the training set, yielding a smaller dataset and consequently a less

TABLE 3.9. Social metrics yield better performing predictive models for developer status across most projects. Mean AUROC values over 250 runs using stratified sampling for each project. Italicized values indicate models that include an insignificant variable in the explanatory model. Values in bold are the highest mean AUROC value over all models that remained significant after a post-hoc pairwise Wilcox test out of all explanatory models with significant variables. Projects that do not have a value in bold were statistically indistinguishable.

Project	Patches	Patches + Messages	Messages	Messages + Threads
Ant	0.71	0.87	0.87	**0.89**
Axis2_c	0.83	0.84	0.83	*0.82*
Log4j	0.30	0.75	**0.91**	*0.88*
Lucene	0.84	**0.85**	0.83	*0.84*
Pluto	0.79	*0.77*	0.76	*0.74*
Solr	0.76	0.90	0.91	**0.96**

TABLE 3.10. Number of messages is a statistically significant predictor with as little as only one month of data. Stability of models with log of number of messages, for 1 to 6 months. Models using a two or three months time window are slightly more stable across all projects

	Ant	Axis2_c	Log4j	Lucene	Pluto	Solr
(Intercept)	**-3.04**	**-2.97**	**-2.94**	0.55	-1.43	0.42
Messages in 1 month	**1.09**	**0.73**	**1.43**	**0.72**	0.49	0.48
Project age	**-0.34**	-0.27	**-0.45**	**-0.81**	**-0.36**	**-0.84**
(Intercept)	**-3.63**	**-3.64**	**-5.8**	-0.09	**-1.8**	-0.92
Messages in 2 months	**1.2**	**0.83**	**1.63**	**0.87**	0.46	**0.83**
Project age	-0.32	-0.24	-0.17	**-0.79**	**-0.31**	**-0.78**
(Intercept)	**-4.15**	**-3.77**	**-6.3**	-0.25	**-2.24**	-2.64
Messages in 3 months	**1.24**	**0.81**	**1.76**	**1.02**	**0.84**	**1.11**
Project age	-0.29	-0.26	-0.17	**-0.83**	**-0.38**	**-0.67**
(Intercept)	**-4.9**	**-3.27**	**-6.75**	-0.83	**-3.24**	**-3.13**
Messages in 4 months	**1.4**	**0.68**	**1.77**	**1.13**	**0.91**	**1.31**
Project age	-0.24	-0.32	-0.15	**-0.81**	-0.27	**-0.72**
(Intercept)	**-6.05**	**-3.51**	**-7.74**	-0.92	**-4.15**	**-3.37**
Messages in 5 months	**1.42**	**0.7**	**1.86**	**1.13**	**0.96**	**1.32**
Project age	-0.1	-0.32	-0.1	**-0.82**	-0.18	**-0.72**
(Intercept)	**-6.73**	**-3.48**	**-7.78**	-0.98	**-4.46**	**-3.57**
Messages in 6 months	**1.4**	**0.69**	**1.8**	**1.1**	**1.07**	**1.41**
Project age	0	-0.34	-0.08	**-0.82**	-0.19	**-0.77**

reliable model. The median time to become a developer in our OSS projects ranges from 8 months to almost a year (except for Log4j which is 4 months). Choosing time windows of 1 month to 6 months for observing participants' activity will still include more than half of the developers in the dataset.

The modeling results can be seen in Table 3.10. For one or two months the models are not as significant as other models. But afterwards all the models are statistically significant, valid, and surprisingly stable.

Model stability only tells one part of the story, *viz.*, it can explain how the model fits the data. However, there is always risk of over-training and evaluation of the predictive power of the models will more effectively demonstrate the value of this model in a realistic setting. We see in Figure 3.6 that the predictive powers of the models differ slightly from one time window to another. Time windows less than 3 months slightly suffer from lower predictive power and time windows of greater than 4 months are almost no better than 3 or 4 months. We choose 3 months as our default because of best overall stability and predictive power (4 months is almost just as good, but with our goal of prediction, the smaller the time window, the better).

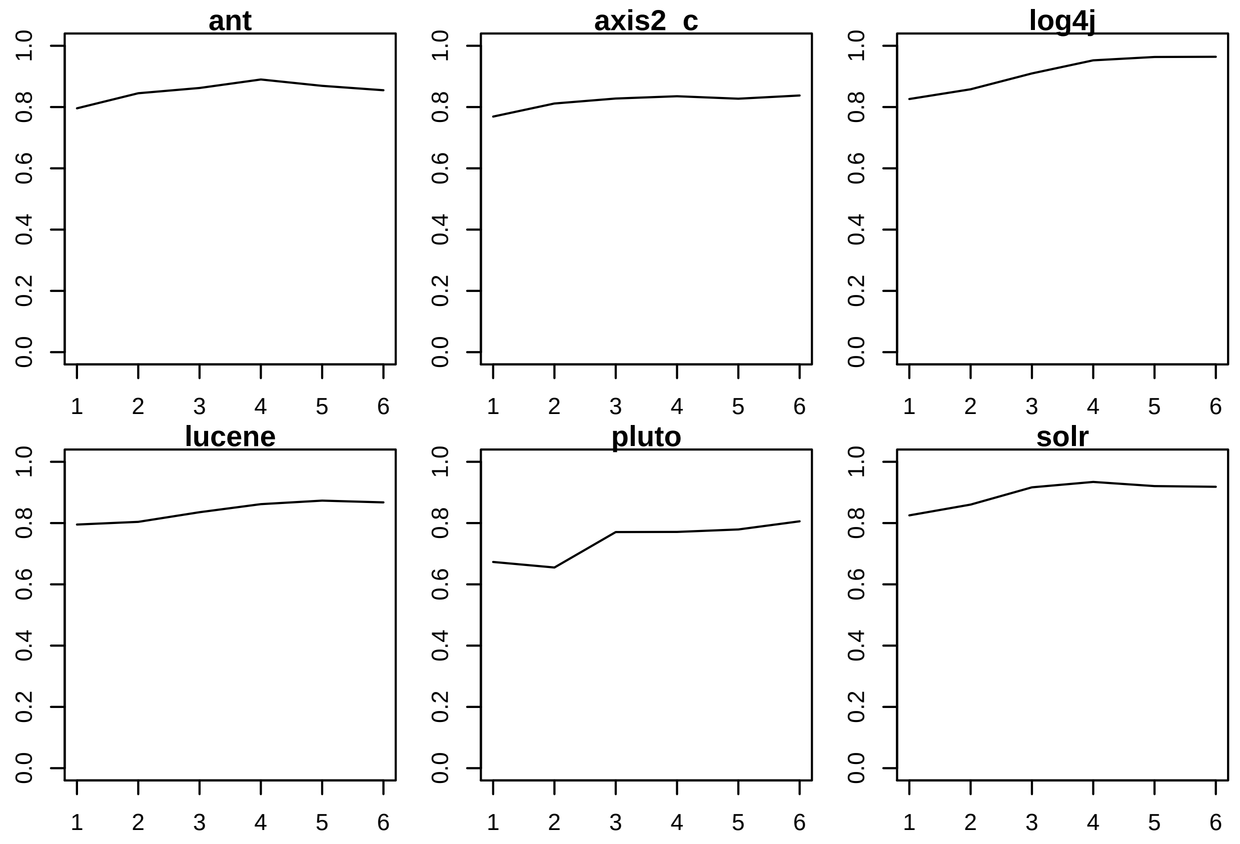


FIGURE 3.6. The predictive power of the model using "number of messages" from 1 to 6 months, each on 250 iterations of modeling using stratified data. The AUROC for each project slightly improves until 3rd or 4th month, and then stabilizes.

Additionally, adding the number of threads to our model improved the prediction results in two projects. Figure 3.5 (bottom) and Figure 3.7 show that adding the number of threads to the model slightly improves prediction performance.

Result 2: *Developer initiation can be modeled with as little as one month's information about the social activity of individuals; using three months yields stronger and more stable result.*

3.5.3. Research Question 3. Previous studies on gamification and goal-directed approach behavior, reviewed in Section 3.2.2, suggest viewing developer initiation as an incentives-based process, in which uninitiated developers are motivated by obtaining committership. In this light, developers may exhibit distinctive dynamics of (social) activity around "reward" (*i.e.,* committership) time, *e.g.,* similar to Stack Overflow users around the time of obtaining a badge [**59**, **60**], or IBM social networking site users around

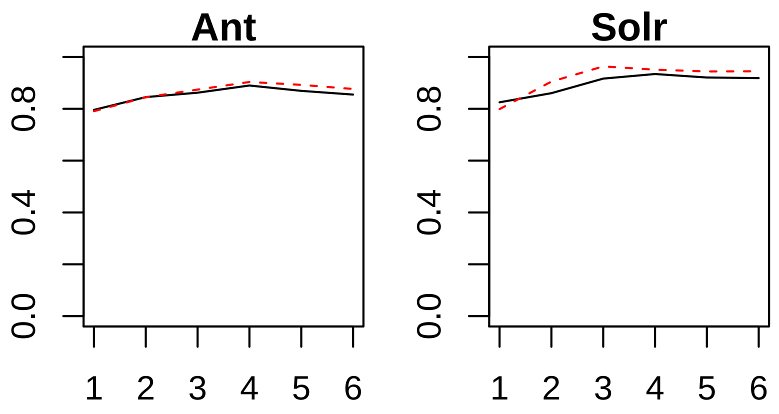


FIGURE 3.7. The AUROC of two projects with two different models. Black lines represent models using only "Number of messages" while red dashed lines represent models with "Number of Threads" added to them. The latter performs slightly better than the former.

the time of levelling up [**58**]: substantial rise in activity before achieving the goal followed by an immediate drop.

We start by visually analysing social activity trends using a six-months window prior to and after developer initiation. After excluding developers who became committers in fewer than six months after their first interaction on the mailing list (*e.g.,* were already committers from the beginning of the project) and developers who became committers later than six months prior to the end data collection date, for which not enough measurements of their social activity exist, our sample consists of 67 developers across the six ASF projects. For each project and periods of six months (approximated as 30 day intervals) prior to and after committership status (centred at time 0), Figure 3.8 depicts boxplots of the number of messages by developers in that project. Visual inspection of Figure 3.8 suggests a peak in social activity close to committership status (either one month before—in Axis2_c—, or one—in Ant, Lucene, or Pluto—or two—in Log4j—months after initiation), followed by a drop in communication in the following months, in all projects except Solr.

TABLE 3.11. Mann-Kendall Trend test for 6 months prior to and after becoming committer.

	6 months before		6 months after	
	tau	p-value	tau	p-value
Ant	0.20	0.00	-0.05	0.27
Axis2_c	0.11	0.18	-0.09	0.17
Log4j	0.28	0.07	-0.08	0.30
Lucene	0.21	0.00	-0.01	0.83
Pluto	0.23	0.03	-0.12	0.08
Solr	-0.01	0.90	0.01	0.89

To statistically evaluate the presence of trends in the data, we split each developer's communication history into two equal-sized groups, at the initiation point, and test for the existence of increasing and decreasing trends prior to and following developer initiation. We test for a simple monotonic trend between time t and the values at time t, $y(t)$, using the non-parametric Mann-Kendall test [**73**]. For 3 of the 6 projects we confirm a statistically significant monotonic trend for the six months prior to developer initiation, while for one of the six we confirm only a statistically suggestive monotonic trend (Table 3.11). On the other hand, for the six months following developer initiation, we cannot confirm a monotonic trend in any of the projects. However, for five of the six projects (all but Solr) the test statistic (tau) is negative, indicating a downward trend.

To gain more insight into these quantitative results, we proceed to cluster the individual patterns of communication activity for the 67 developers in our sample. Since we are interested in trends or patterns of social interaction, using the observed (*i.e.,* unsmoothed) values would introduce excessive noise. Instead, we consider LOESS [**74**] smoothed versions with unit span of each developer's time series of social interactions, to obtain the clearest trends. Then, we perform clustering of the different communication patterns using an open card sorting approach [**75**], where each card contains the communication pattern of each developer. Card sorting is a technique frequently used in qualitative research for categorisation (assigning cards into meaningful groups). In case of open card sorting, there are no predefined groups, but rather groups emerge and evolve during the sorting process. To reduce bias, each of the first three authors independently assigned cards to clusters, after which review and discussion took place until all participants agreed on the final set of clusters. The results are presented in Figure 3.9.

We reflect on a number of observations. First of all, six out of the seven clusters (cluster 6 contains too few developers to draw any meaningful conclusions), comprising the vast majority of developers, fall into either of three groups of patterns. The first group, comprising clusters 1, 2, 3 and, to a lesser extent 5, consists of developers that exhibit the anticipated goal-directed approach behavior: a ramp-up of social activity in the period of time immediately before (cluster 1), right around (cluster 2) and shortly after (cluster 3) developer initiation, followed by a cooling off period. Developers in cluster 5 do not exhibit the drop in social activity after becoming committers.

The second group, consisting of cluster 7, includes developers for which committership does not seem to impact their social behavior. Further manual inspection reveals that developers in this cluster often are well known members of the ASF community (*e.g.,* Apache evangelists, book authors) or committers elsewhere already within the ASF, suggesting that established ASF contributors might need much less reputation-building than newcomers when trying to become committers.

Finally, the third group, consisting of cluster 4, comprises developers that continue to intensify their social activity beyond reaching committer status. Further manual inspection of this cluster suggests other external factors at play, such as incubator projects being "graduated" as sub-projects of Lucene (*e.g.,* Nutch) or Ant (*e.g.,* Ivy), together with the merger of their respective developer mailing lists into the mailing lists of the main projects at the time of committership.

Result 3: *Typically, a ramp-up of social activity occurs in the period of time close to developer initiation. Following a short cooling off period right after initiation, more individualized socialization patterns emerge.*

3.5.4. Research Question 4. We are also interested in understanding how trust evolves over the life of each project. Looking back at previous models, we see that in all cases, the coefficient for "Project Age" is negative, implying it becomes increasingly difficult to become a developer over time. To verify this hypothesis, we replaced "Project Age" with a dummy binary variable called "IsSecond" which is true for the second half of population (sorted in ascending order by their "Project Age") and is false for the first half. If the coefficient of this variable is still negative across projects it will confirm our hypothesis that being initiated a developer becomes increasingly more difficult over time. Ideally "Project Age" should be broken to smaller partitions (4 or more) to give us a higher resolution view, but increasing the resolution would result in even less sample points in each partition, making the results less reliable.

TABLE 3.12. It becomes increasingly more difficult to earn trust in an OSS. Models show a dummy variable "IsSecond" which is true for individuals that $Project\ Age > median(Project\ Age)$. It is seen that joining the project later has a negative effect on one's chance of becoming a developer.

	Ant	Axis2_c	Log4j	Lucene	Pluto	Solr
(Intercept)	**-5.5**	**-4.21**	**-7.69**	**-4.74**	**-2.99**	**-6.59**
Number of patches	**0.62**	**0.59**	-0.76	**0.62**	**0.85**	-0.2
Number of messages	**1.08**	**0.56**	**1.89**	**0.72**	0.49	**1.17**
IsSecond	-0.36	**-2.08**	-0.9	**-2.46**	**-2.1**	-1.34
(Intercept)	**-5.76**	**-4.93**	**-7.04**	**-5.42**	**-3.8**	**-6.33**
Number of messages	**1.24**	**0.82**	**1.78**	**0.99**	**0.88**	**1.07**
IsSecond	-0.57	**-1.84**	-0.99	**-2.67**	**-2.01**	-1.29

Two different logistic regression models were fit to the data, and the models are given in Table 3.12. We see that for all projects, the coefficient of "IsSecond" is negative and statistically significant across three of the six projects. In a fourth project, *solr*, the coefficient is statistically suggestive at the 10% level in both models ($p = 0.103, 0.108$, respectively). Statistically insignificance, does not imply the converse, we can only state that the result is "inconclusive". The negative sign of the coefficient, however, indicates a negative skew in the confidence intervals about the predictor. In most projects "age" has a negative effect on chances of becoming a developer. Based on these observations, we conclude:

Result 4: *It becomes more difficult for individuals to become committers as the project matures; late stage developers may have to put more effort to gain the same level of trust.*

3.6. Conclusion and Threats to Validity

We presented strong evidence for the determining role that social networking activities play in becoming a developer in the studied ASF project. Surprisingly, to this end, social communications are a better predictor than patching activity. We also present evidence that developers' early social activities in the project identify them as such. Moreover, prior to becoming developers, participants typically ramp-up their socialization activities, in anticipation of their upgraded status. Expectedly, we also find that community trust is more difficult to attain with time as the community likely takes longer to identify trustworthy contributors.

Our methods are based solely on two-way social links representing messages sent between project participants, but is oblivious to the content of those messages. Clearly, knowing the content of the emails would add another layer of information that can be mined. However, the quality of our predictions while disregarding content is an indication of the strong influence of the social link structure. This may be of independent interest to the management and security communities.

Our results in no way imply causality, rather a strong statistical correlation between the measured attributes that can be used for prediction and further research.

We recognize several threats to the validity of our approach and conclusions. The dataset gathered here was from 6 projects, all from the Apache Software Foundation. This might impose a limitation on the pattern of communication and contribution in these projects that will limit the applicability of our results to other OSS projects. It also may be that there is a systematic bias in our data, meaning what we measure is not the likelihood of obtaining developer status, *e.g.,* people may be assigned to be developers (rather than being chosen) and are using ESNs to familiarize themselves with the community. Although this assumption is quite contrary to ASF's guidelines[7], it is not hard to imagine other scenarios where developers are not chosen as we think they are.

When looking for trends in the socialization data before and after one becomes a developer, we are obviously limited by the number of committers with 12 month email history that is available in our dataset, which is 67 committers after eliminating insufficient data entries. In addition, the analysis of short time-series data is in general susceptible to noise arising from the multiple assumptions used, which in our case is mitigated to an extent by our relatively straight forward analysis.

Having more projects is desirable, but practically, we had to select projects with a large number of developers for the predictive models to have reasonable statistical power. We cannot address private communication between developers which may impact the structure of the social network. This limitation, however, affects all such work of this nature and we do not believe that it severely limits the usefulness of our results.

[7]`http://www.apache.org/foundation/faq.html`

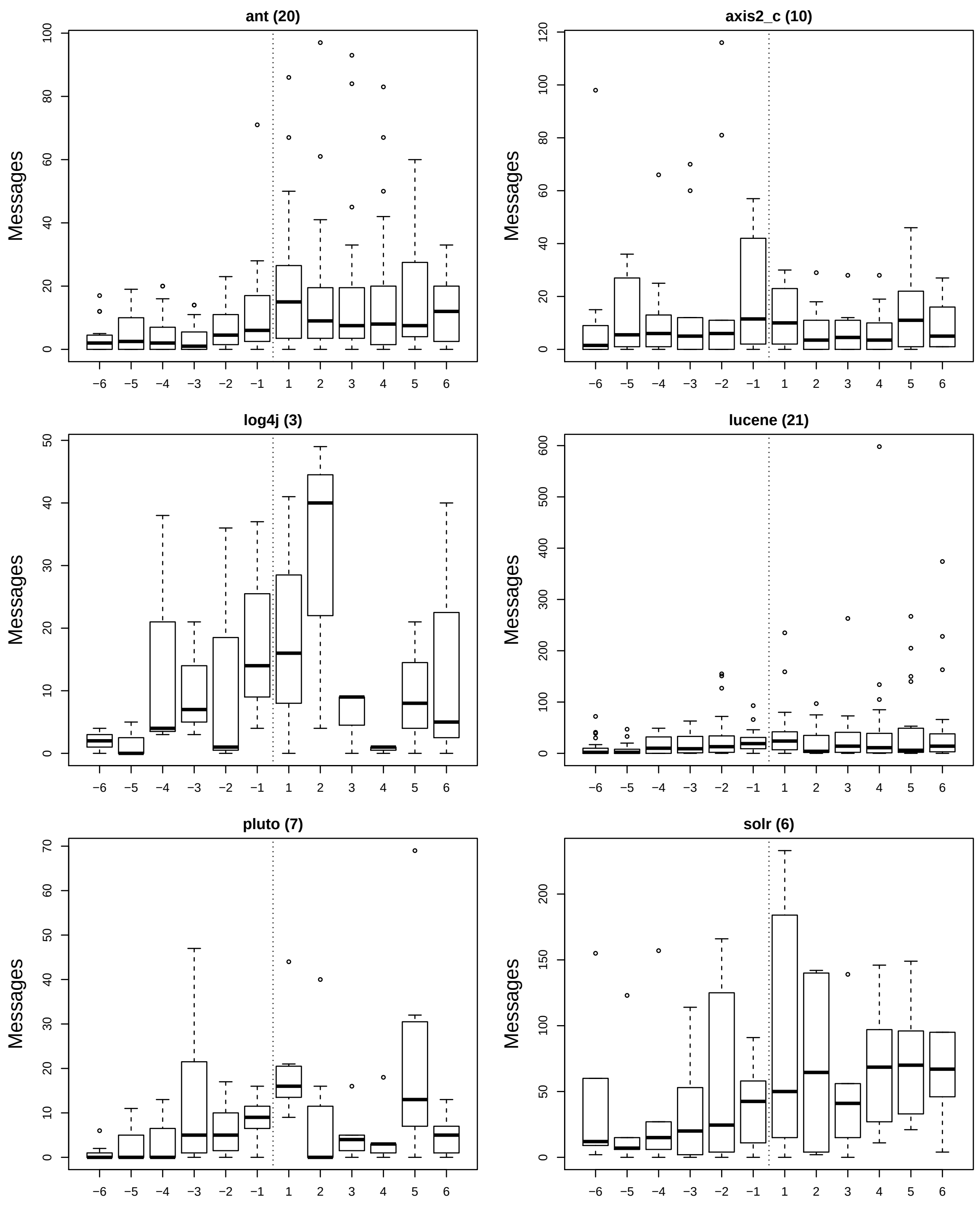


FIGURE 3.8. Boxplots of individual's communication activity around time to become a committer. The numbers in parentheses besides each project's name indicate the number of individuals in the boxplots. Developers who became committers in fewer than 6 months after their first message on the mailing list (*e.g.,* were already committers from the beginning of the project) and developers who became committers later than 6 months prior to the end data collection date, for which not enough measurements of their social activity exist, were excluded.

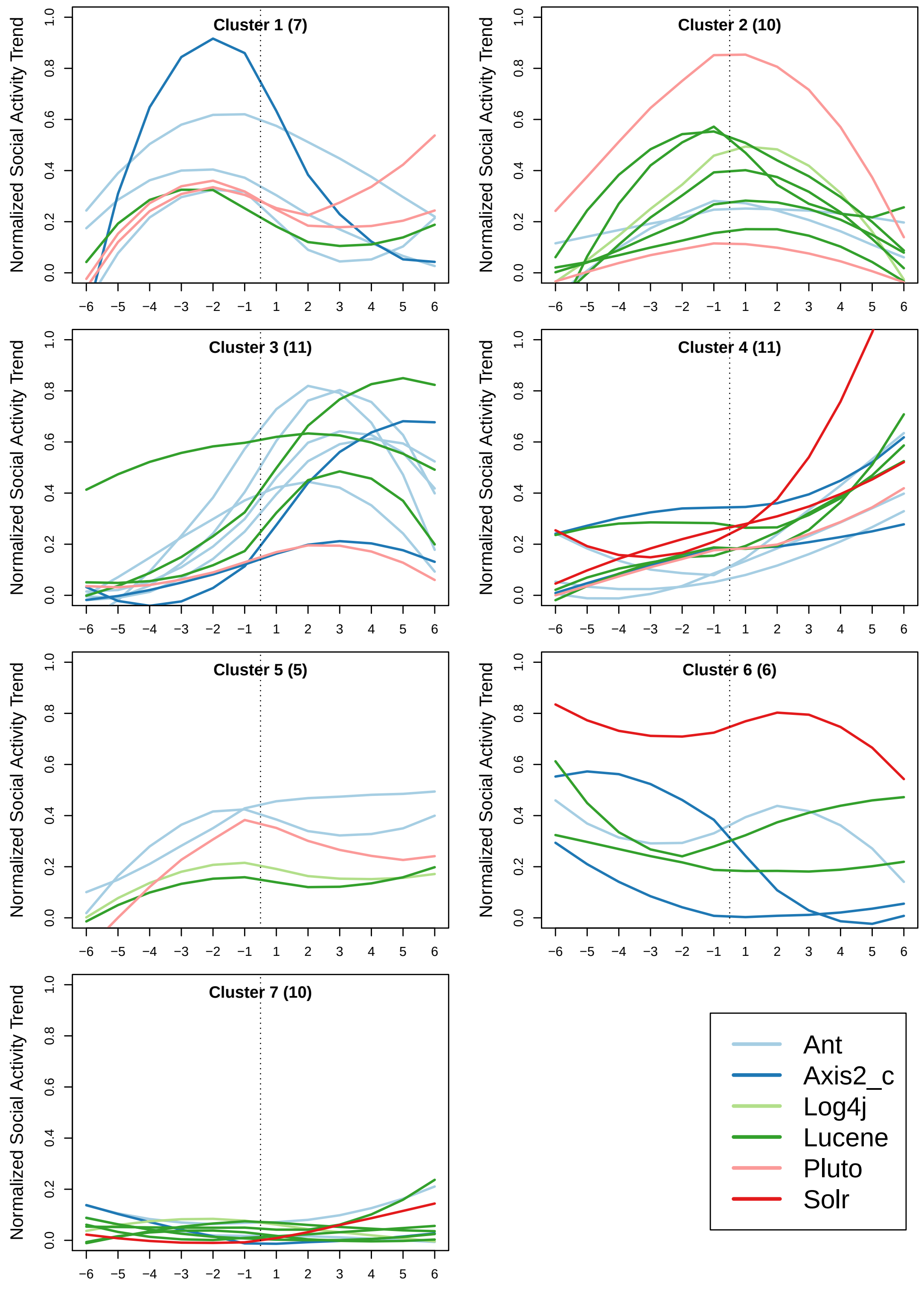


FIGURE 3.9. Patterns of social activity prior to and after developer initiation, as resulting from manual clustering based on open card sorting. Developers from the same project are assigned the same color. The numbers in parentheses besides each cluster number indicate the number of individuals in that cluster.

CHAPTER 4

Measuring the Effect of Social Communications on Individual Working Rhythms: A Case Study of Open Source Software

This chapter was in most part completed by others as part of the mentioned research paper. My contributions consisted of data gathering, parsing and sanitization. I include the whole chapter here for completeness.

4.1. Introduction

Recently, much attention has been paid to social networks [**37**, **76**, **77**], where nodes represent different individuals and links between pairs of nodes mean the corresponding individuals are friends or directly communicate with each other. Generally, there are two reasons that can explain this great interest in networks. First, can conveniently model the topological structure of complex systems, providing a series of structural characteristics [**78**, **79**] that can help differentiate the roles of individuals working on these systems. For example, Albert *et al*. [**80**] and Holme *et al*. [**81**] found that the communication efficiencies of networks are more likely to depend on the nodes with larger degrees or betweenness centralities, *i.e*., the network performance decreases quickly once these nodes are removed. Second, it is widely believed that network structure, or more specifically, communications themselves can influence the individual actions, and thus some cooperative behaviors such as synchronization [**82**–**84**] naturally emerge.

However, such influence is often difficult to measure, and subject to conflicting viewpoints. Consider the increasing interest of developers to contribute to open source software (OSS) projects [**16**, **85**, **86**]. Developers communicate through emails and commit to different files in real time. Since most work on OSS is voluntary, the success of these OSS projects is mainly determined by the committing activities of developers [**87**], which leads to an important question [**88**] in this area: whether and how do the communication activities influence the committing activities? For this question, different people may have different answers before such influence can be quantified.

It is often argued that social communication activities delay programming activities, since both of these activities may compete for the time resources of developers. As a result, people always prefer to find ways to reduce "communication overheads". Baldwin and Clark [**89**] argued that the communication and coordination in large systems can be significantly reduced by adopting proper design rules. On the other hand, in our society, productivity and communication often go hand-in-hand. In OSS projects, it can be argued that once a user finds a bug [**90**, **91**], she would want to report them as soon as possible in order to gain some sense of achievement, while once the developers received some (possibly negative) evaluations of their work, they may respond by updating the software right away, in order to preserve their reputation among the social circle of developers. In such a situation, since task-relevant information flows through communication activities, it can be argued that communication activities accelerate committing activities in the OSS projects to some extent.

Then, which is it? Do the communication activities impede committing activities or do they accelerate them? Although a recent study [**41**] on OSS projects indicates that there are strong correlations between the number of messages sent by an individual and the number of code changes he/she made, this result still cannot answer the above question. Clearly, both of these activities are positively correlated with the time interval over which they take place; however, this result doesn't necessarily imply that one accelerates the other.

In order to answer this question more definitely, here we first provide the definition of working rhythm, or committing rhythm more specifically, for developers. Then, we propose two methods to measure the effects of communication activities on committing rhythms. In particular, the main contributions of this paper include the following two parts:

(1) *Macroscopic view*. We build communication networks for code developers from ten years of email records in 31 OSS projects, and utilize the local structural properties in complex network theory, such as outgoing degree and incoming degree, to quantify the social status of developers. We find that the developers with higher social status, represented by the nodes with larger number of outgoing or incoming links, always have faster working rhythms and thus contribute more per unit time to the projects.

(2) *Microscopic view*. We introduce multi-activity time-series and propose the definitions of *evaluation* and *response* latencies between the successive committing and incoming communication activities in order to quantitatively measure the dependency between work and talk activities. We

introduce a mechanism to generate independent simulated time-series of incoming communication activities which have precisely the same statistical properties as the real ones. By comparing measurements on the simulated time-series with those on the real ones, we find that the committing and communication activities may significantly accelerate each other in OSS systems.

We study how work and talk activities interact; our findings suggest that frequent, interleaving communication around committing activities is essential for effective software development in a distributed setting; but our findings may have broader implications beyond OSS. Many real-world systems can be described by complex networks and individual actions can be modeled using time-series, the methods proposed here can also be used to quantify certain relationships in other areas.

The rest of the paper is organized as follows. In Section 4.2, the communication and committing data obtained from OSS projects are briefly introduced, where communication networks and committing networks are constructed and some basic properties are provided. In Section 4.3, the methodologies, including the definition of committing rhythm and the network and time-series based methods are introduced. In Section 4.4, the main results are obtained based on the proposed definition and methods. The paper is finally concluded in Section 4.5.

4.2. Data Description

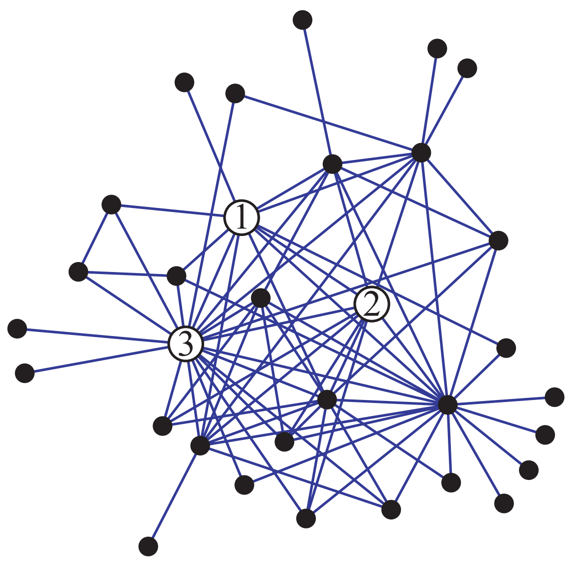
A total of 31 OSS projects were obtained from the *Apache Software Foundation* on March 24th, 2012. For each project, a communication social network is constructed from online developer mailing lists [**92**]. These mailing lists are used for communication and coordination among the normal users and developers, where each email has an ID, a sender ID, and a reference ID with date time and body. Here, the reference ID is the ID of the email that this email is in response to. In such a network, the nodes are the people sending messages on the list, if a person P_1 replies to a message from another person P_2, then there is a directed link from the node representing P_1 to that representing P_2. The social networks constructed by this method are directed networks, and each node v_i in a network will have an incoming degree k_i^{in} and an outgoing degree k_i^{out} representing the numbers of directed links from and to this node, respectively. The average outgoing degree of the network is denoted by $\langle k_{out} \rangle$, and the average incoming degree of the network has the exactly same value. Meanwhile, the committing activities of developers on different files in each project are gathered from the corresponding Git repository and can also be considered as a network, where a node represents a developer or a file, and a link between a developer and a file represents that the developer

TABLE 4.1. Some basic properties of the 31 OSS projects.

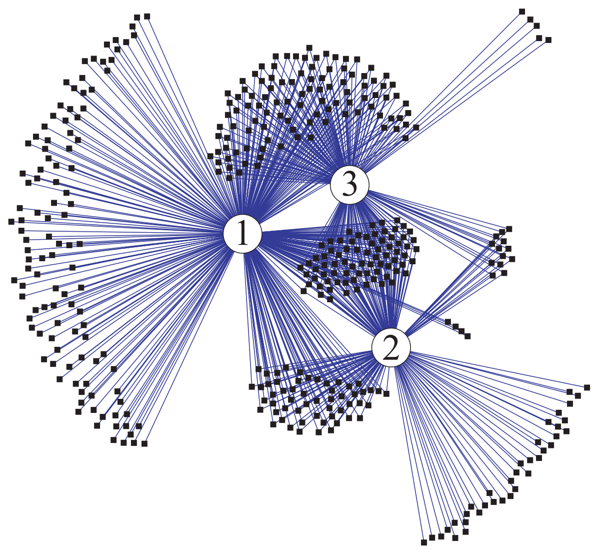
Project	N_T	N_D	N_F	$\langle k_{out} \rangle$	$\langle d_F \rangle$
Accumulo	75	5	1622	3.5	833.2
Mahout	552	16	5123	4.4	698.9
Lucene	2148	41	6674	4.2	414.2
Nutch	862	16	3072	3.4	424.3
Derby	1128	35	6563	5.6	660.1
Ode	377	18	11006	3.8	1013.4
Openejb	179	38	43960	5.8	2374.0
Log4php	88	9	1409	2.1	242.0
Wicket	540	24	48045	5.5	3907.3
Log4j	540	19	5519	2.3	472.7
Bookkeeper	32	3	407	3.0	245.0
Xerces2_j	922	33	3732	1.7	347.4
Hive	321	18	7333	3.4	887.2
Axis2_java	3758	76	129978	2.8	3034.1
Hadoop_hdfs	264	25	1153	2.4	171.2
Camel	844	31	36965	3.2	1713.1
Avro	284	12	3021	3.0	387.6
Abdera	196	13	3193	2.7	352.4
Cassandra	397	13	17125	3.7	1534.7
Activemq	2053	29	16788	2.2	946.3
Cxf	443	45	37867	4.2	1726.2
Log4net	297	7	1060	1.4	320.9
Ant	1406	45	11620	3.2	666.5
Empire_db	8	5	2341	0.88	807.6
Axis2_c	608	24	10262	4.4	805.3
Cayenne	170	20	31489	3.8	2629.9
Log4cxx	92	6	2966	1.7	730
Harmony	709	25	14898	9.0	636.8
Pluto	265	23	5971	3.1	483.0
Solr	839	19	8534	3.76	655.8
Ivy	68	9	3513	2.3	523.2

has contributed to the file (i.e., add some codes in this file). The committing networks constructed by this method are bipartite networks, i.e., links only exist from developers to files. A developer v_i^D has a degree d_i^F representing the number of files s/he has committed. Then the average degree $\langle d_F \rangle$ in a project denotes the mean number of files committed by a developer. Note that, in these projects, users can have multiple aliases; these were resolved using a semi-automatic approach devised by Bird *et al.* [**41**].

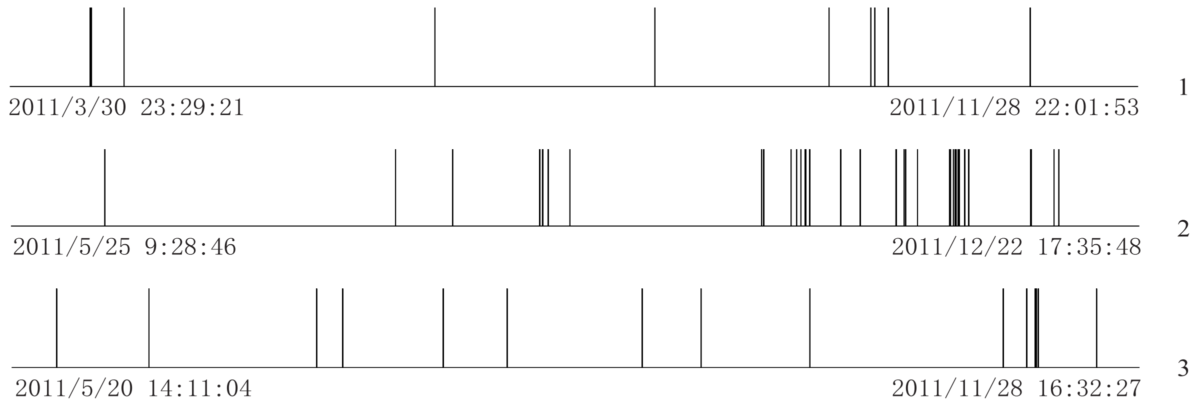
Several basic properties, including name of the project, number of users N_T (including the developers), number of developers N_D, number of files N_F, average outgoing degree $\langle k_{out} \rangle$ in the corresponding communication network, and average committing files $\langle d_F \rangle$, of the 31 OSS projects are presented in TABLE



(a) Social network



(b) Committing network



(c) Time series of committing activities

FIGURE 4.1. The topological structure of (a) social network and (b) committing network of the project called *bookkeeper*, where the three marked unfilled nodes are the developers who contributed to the project and the filled nodes represent the other users and files in the corresponding networks. (c) The time-series of the committing activities of the three corresponding developers, with the time of their first and last committing activities provided, where the horizontal axis denotes time (in one second) and each vertical line corresponds to an activity of adding codes.

4.1. By considering all the projects together, there are totally 20465 users, 702 developers and 483,209 files. In particular, the social network structure and the committing network structure of the project called *bookkeeper* are shown in Fig. 4.1 (a) and (b), respectively, where the three developers are represented by the unfilled nodes and marked by 1, 2, and 3, respectively, while the other users and files are represented by the filled nodes, in the corresponding networks. Meanwhile, the time-series of committing activities of the three corresponding developers are visualized in Fig. 4.1 (c), where there is a vertical line if the developer added codes at that time.

4.3. Methodology

4.3.1. Definition of committing rhythm . Suppose there are totally $M(M \geq 2)$ activities for an individual at different consecutive time $t_1, t_2, \ldots, t_M$ satisfying $t_1 < t_2 < \ldots < t_M$, there will be $M-1$ inter-activity time buckets, denoted by

$$\Delta t_i = t_{i+1} - t_i, i = 1, 2, \ldots, M-1. \tag{4.1}$$

Since many human activities are Poisson processes [**93**] where independent events occur at a constant rate λ, the inter-activity time between two consecutive activities of an individual follows an exponential distribution as follows:

$$P(\Delta t) = \lambda e^{-\lambda \Delta t}. \tag{4.2}$$

Committing activity is no exception either, as shown in Fig. 4.2, where the near linear functions in semilog coordinates indicate that both inter-activity time distributions of communication and committing activities follow exponential distributions.

Therefore, the average inter-activity time of an individual, calculated by

$$\langle \Delta t \rangle = \frac{\sum_{i=1}^{M-1} \Delta t_i}{M-1} = \frac{t_M - t_1}{M-1}, \tag{4.3}$$

is close to the inverse of the only parameter λ of the distribution, and thus is a reasonable way to measure its committing rhythm, i.e., smaller average inter-activity time means faster rhythm.

4.3.2. Network based method . Generally, communications between individuals can be described by social networks, as shown in Fig. 4.1 (a), while in the area of social network analysis, the status of an individual is more likely to be characterized by its local structural properties [**78**], such as degree [**79**], rather than the number of communication records themselves. In particular, the social status of a developer here is characterized by the incoming and outgoing degree of the corresponding node in the network, since in OSS projects, it is considered that a developer with higher incoming degree is widely trusted in the local society while one with higher outgoing degree has a better sense of responsibility for the project. In most cases,

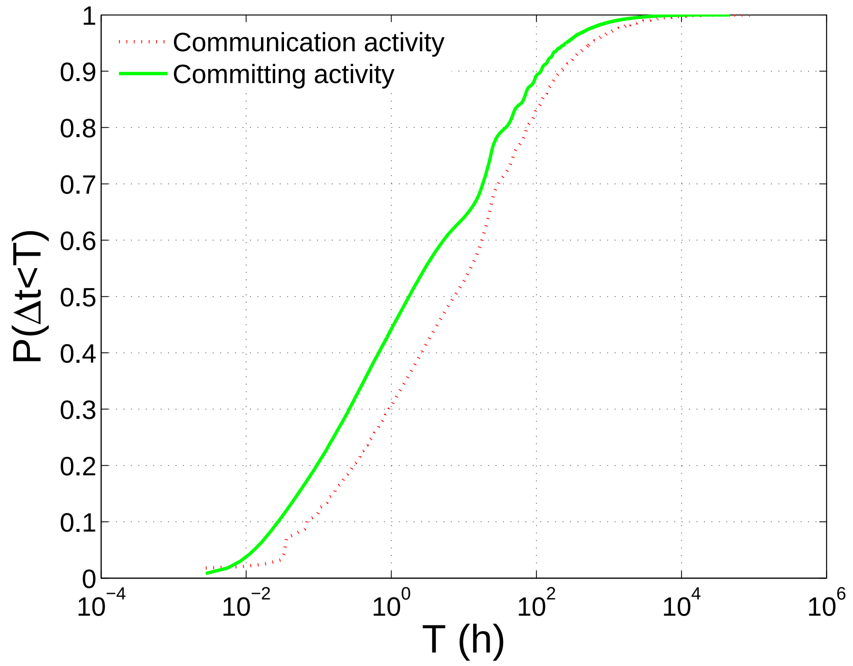


FIGURE 4.2. The cumulative inter-activity time distributions of the communication activities and the committing activities in 31 OSS projects.

these two local structural properties are correlated with each other to a certain extent, i.e., an individual with higher responsibility for the project is always widely trusted by others in the corresponding local society.

In order to study such social responsibility on committing rhythms of developers, it is intuitive to divide the developers into different groups by their incoming or outgoing degrees. Since there are only 702 developers, here the developers are only divided into two groups in order to make sure that the final results have statistical meaning. In particular, there are two cases as follows:

- *Case I*: The developers are divided into two groups according to their incoming degrees, i.e., the developers with incoming degree $k_{in} \leq 50$ and the developers with incoming degree $k_{in} > 50$.
- *Case II*: The developers are divided into two groups according to their outgoing degrees, i.e., the developers with outgoing degree $k_{out} \leq 50$ and the developers with outgoing degree $k_{out} > 50$.

Note that here the degree threshold value to divide the developers into two groups is set to 50, which is just because, in such a case, there are similar number of activities in the two groups, although, in fact, the results almost keep the same for different degree threshold values. Then, all the inter-activity time buckets of the developers in the same group are put together and is compared with those of the developers in the other group. If committing activities are driven by social responsibility, it is reasonable to find that,

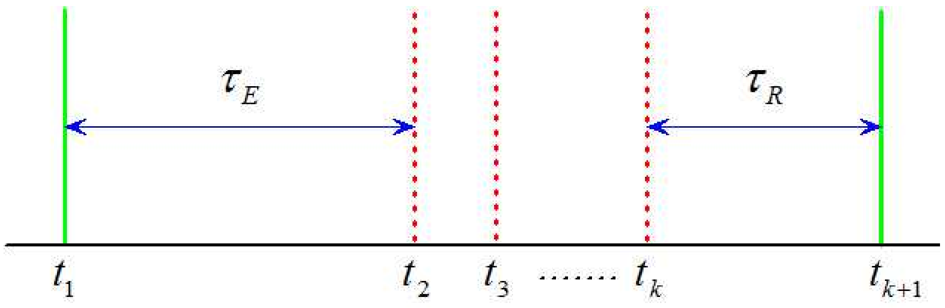


FIGURE 4.3. The definitions of the evaluation latency τ_E and the response latency τ_R for a developer. The horizontal axis denotes time. Each solid vertical line corresponds to a commiting activity of adding codes at time t_1 and t_k, while each dotted vertical line corresponds its incoming communication activities, *i.e.*, he/she received emails at time $t_2, t_3, \ldots, t_k$ satisfying $t_1 < t_2 \leq t_3 \leq, \ldots, \leq t_k < t_{k+1}$. Then the evaluation latency τ_E and the response latency τ_R are defined by Eqs. (4.4) and (4.5), respectively.

statistically, the developers with larger incoming or outgoing degrees have faster committing rhythms. Otherwise, if communication and committing activities compete the time resources of developers, the opposite phenomenon may be observed.

4.3.3. Time-series based method . At a fine-grained level, the interaction between communication and committing activities can be directly measured by comparing their time-series. In fact, there is an evaluation-response mechanism in many OSS projects. That is, once a developer $\mathcal{D}$ submits a section of codes at time t_1, denoted by action A_1, s/he may receive several evaluations (such as code reviews or problem reports) from other users (including other developers) at time $t_2, t_3, \ldots, t_k$ via email. $\mathcal{D}$ may respond to these evaluations by adding new section of codes at time t_{k+1}, denoted by action A_2. Suppose that actions A_1 and A_2 are successive, *i.e.,* $\mathcal{D}$ does not add any code in the time interval (t_1, t_2), and we have that $t_1 < t_2 \leq t_3 \leq, \ldots, \leq t_k < t_{k+1}$, the evaluation latency and the response latency then are defined by

$$\tau_E = t_2 - t_1, \tag{4.4}$$

$$\tau_R = t_{k+1} - t_k, \tag{4.5}$$

respectively, as shown in Fig. 4.3.

Using the actual incoming communication activities of each individual, but assuming that there is no coordination between communication and committing activities, it is possible to generate a series of simulated incoming communication activities for each individual independently. In particular, for each developer v_i,

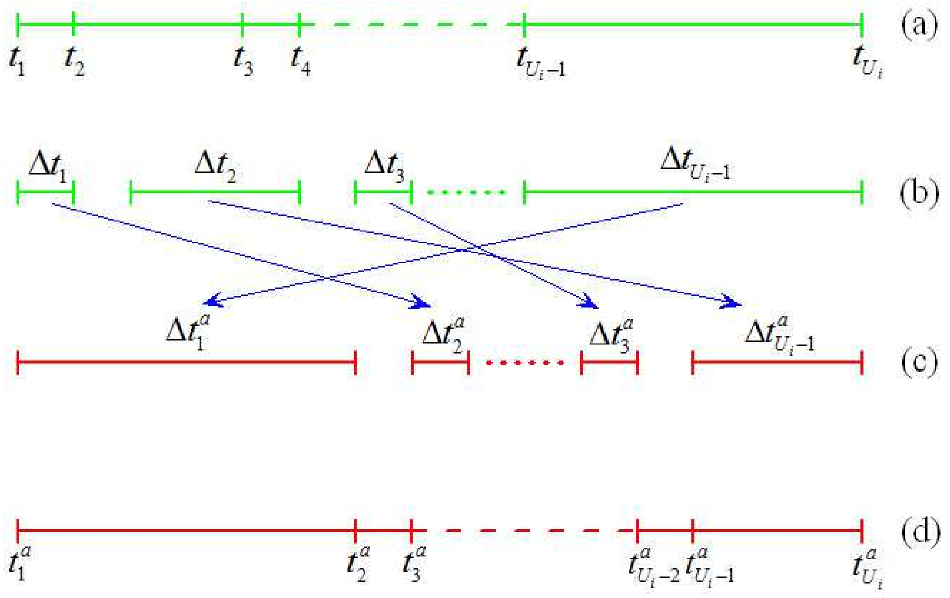


FIGURE 4.4. The steps to generate a simulated time-series of incoming communication activities for each developer.

the precise method to generate such a simulated time-series of incoming communication activities is as follows:

(1) Suppose there are totally $U_i (U_i \geq 2)$ incoming communication activities for v_i at different times, denoted by $t_1, t_2, \ldots, t_{U_i}$, respectively, as shown in Fig. 4.4 (a). Then, $U_i - 1$ ordered time intervals, denoted by $\Delta t_1, \Delta t_2, \ldots, \Delta t_{U_i-1}$, respectively, can be obtained by $\Delta t_i = t_{i+1} - t_i$, as shown in Fig. 4.4 (b).

(2) Randomly rearrange the $U_i - 1$ time intervals and get a new sequence of time intervals, denoted by $\Delta t_1^a, \Delta t_2^a, \ldots, \Delta t_{U_i-1}^a$, respectively, as shown in Fig. 4.4 (c). This essentially generates random orderings of "idling periods" for the developer, but ensures that his "idling" periods are exactly the same as actually observed.

(3) Weld these new ordered time intervals one by one, as shown in Fig. 4.4 (d), and then get a new time-series $t_1^a, t_2^a, \ldots, t_{U_i}^a$, satisfying that

$$\begin{cases} t_i^a = t_i, i = 1, \\ t_i^a = t_{i-1}^a + \Delta t_{i-1}^a, i \geq 2. \end{cases} \tag{4.6}$$

Note that the simulated time-series of incoming communication activities generated by this mechanism preserves similar statistical properties as the real one, *viz.,* the same distribution of inter-activity interval time.

By replacing the real time-series of communication activities by this simulated time-series and comparing with the real time-series of committing activities, we can get two series of simulated evaluation and response latencies which are still calculated by Eqs. (4.4) and (4.5), respectively. Generally, for the real communication and committing activities, there are several possible relationships between them, which are listed as follows and can be characterized by comparing the real and simulated evaluation and response latencies.

(1) *They are independent from each other.* If this is the case, the distributions of real evaluation and response latencies will be statistically indistinguishable from the simulated ones.

(2) *They delay each other.* In reality, both communication and committing activities do take time, *i.e.*, they compete for the time resources of developers. In addition, developers may spend time responding to bug reports, questions, challenges *etc* in the emails; this might delay committing activities to a certain extent. If this is the case, the actual evaluation and response latencies will be statistically longer than the simulated ones.

(3) *They accelerate each other.* As discussed earlier, the desire to enhance and or maintain reputations may incentivize users to respond more quickly to tasks that relate to received email correspondence. Bug finders may accelerate bug reports to gain recognition. Likewise, programmers may be hastening to respond to bug reports, or design/coding critiques, in order to maintain their peer-reputation. If this is the case, statistically, the real evaluation and response latencies will be relatively shorter than the simulated ones.

Certainly, there are also other cases where only one kind of activities influence the other. For example, only incoming communication activities accelerate committing activities. In this case, it can be expected that, statistically, the real response latencies will be relatively shorter than while the real evaluation latencies will be close to the simulated ones. And other cases can be confirmed by the similar manner.

4.4. Results

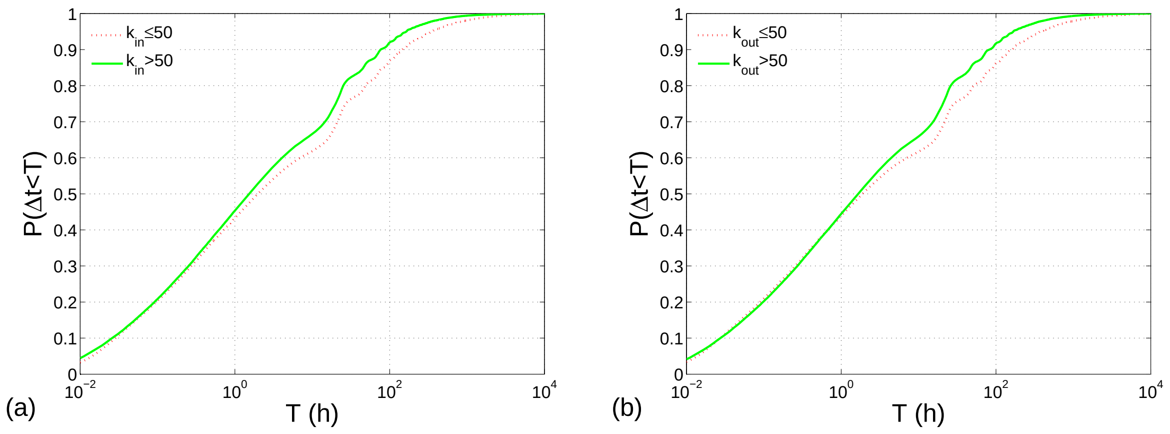


FIGURE 4.5. The cumulative inter-activity time distributions of the committing activities for different groups of developers characterized by their (a) incoming degrees and (b) outgoing degrees. As one can see, here the difference of cumulative inter-activity time distribution between two distinct groups of developers is mainly introduced when $\Delta t > 1$ (h).

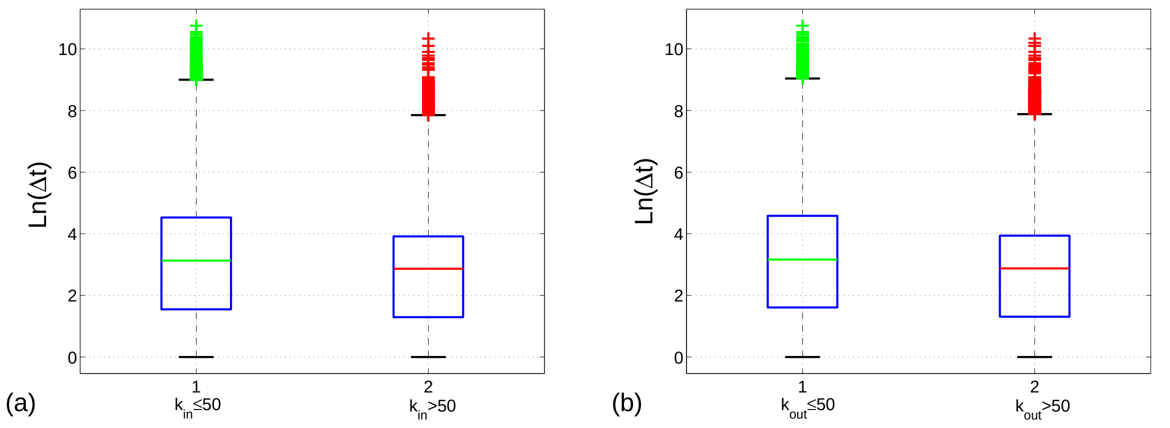


FIGURE 4.6. The corresponding box-and-whisker diagrams of the inter-activity time for different groups of developers characterized by their (a) incoming degrees and (b) outgoing degrees. Here, only the inter-activity time with length longer than one hour is considered, and the y-axes are logarithmically transformed in order to present the difference of the committing activities between different groups of developers more clearly.

4.4.1. Higher social status indicate faster committing rhythms . Here, we use network-based measures. As one can see, the human activity rhythms can be statistically analyzed using the cumulative inter-activity time distribution, which allows us to compare the committing rhythms of developers belonging to different groups. The cumulative inter-activity time distributions of the committing activities for different groups of developers characterized by their incoming degrees (*Case I*) and outgoing degrees (*Case II*) are shown in Fig. 4.5 (a) and (b), respectively. By comparison, there are more short inter-activity time intervals for the developers with higher social status, *i.e.*, larger incoming or outgoing degrees, which indicates that, statistically, the developers with higher social status may have faster committing rhythms. More interestingly, it is found that the difference of cumulative inter-activity time distribution between the two distinct groups of developers is mainly introduced when $\Delta t > 1$ (h) while there seems no difference between them when $\Delta t \leq 1$ (h), which suggests that social status can only influence the long-term, but have little effects on the short-term committing rhythms. This is reasonable because the short-term committing rhythm is more a reflection of the developer's programming habit, *i.e.*, saving codes in real time when he/she commits files, which is for sure seldom influenced by social factors. Just considering the inter-activity time with length longer than one hour, the differences of committing rhythms between the developers belonging to different groups can be presented by the box-and-whisker diagrams more clearly, as shown in Fig. 4.6 (a) and (b), respectively.

In order to provide more credible results, simple T-tests are implemented for both cases and the statistics including the average inter-activity time length, T-value, and significance are recorded in TABLE 4.2, where k means k_{in} for *Case I* and k_{out} for *Case II*. Generally, the average inter-activity time length of the developers with larger incoming (or outgoing) degrees is much smaller than that of the developers with smaller incoming (or outgoing) degrees, and the differences in both cases are quite significant, with the relatively large T-values 20.4 and 22.3, respectively. More specifically, denote by Δt_L^{in} (or Δt_L^{out}) and Δt_S^{in} (or Δt_S^{out}) the average inter-activity time lengths of the developers with incoming (or outgoing) degree larger and smaller than 50, respectively. Then, on average, the differences of committing rhythm between the different groups of developers in two cases can be qualified by

$$\Delta t_S^{in} - \Delta t_L^{in} = 57.5(h), \tag{4.7}$$

$$\Delta t_S^{out} - \Delta t_L^{out} = 62.6(h), \tag{4.8}$$

respectively. These differences will be further enlarged if only the inter-activity time buckets with length longer than one hour are considered. For comparison, extra T-tests are also implemented for the two cases in this situation, and the statistics are recorded in TABLE 4.3, where one can see that the gaps are almost doubled when only considering long-term rhythms.

Moreover, From Eqs. (4.7) and (4.8), one can see that, by comparison, the social status characterized by outgoing degrees have slightly more remarkable effects than those characterized by incoming degrees on

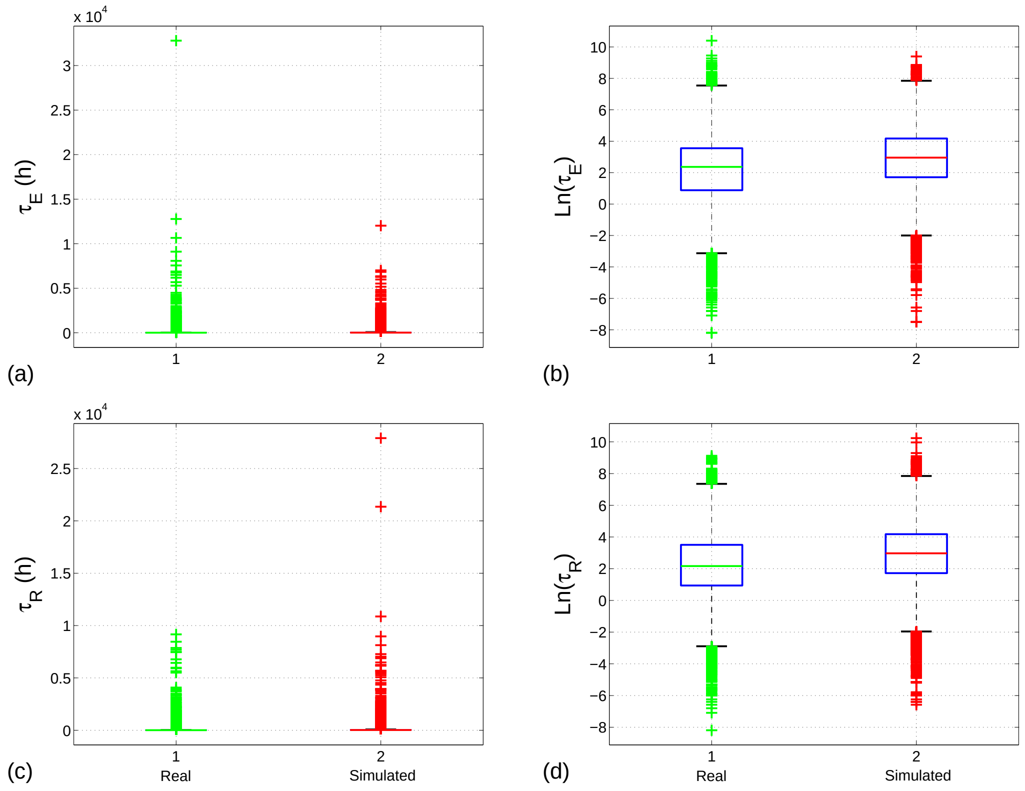


FIGURE 4.7. The box-and-whisker diagrams for real and simulated (a)-(b) evaluation latencies and (c)-(d) response latencies. Here, for (b) and (d), the y-axes are logarithmically transformed in order to present the differences of latencies between real and simulated cases more clearly.

TABLE 4.2. T-tests for the differences of inter-activity time lengths between different groups of developers in *Case I* and *Case II*.

T-test	$k \leq 50$	$k > 50$	T-value	Significance
Case I	103.0 (h)	45.5 (h)	20.4	$p < 10^{-6}$
Case II	111.2 (h)	48.5 (h)	22.3	$p < 10^{-6}$

TABLE 4.3. T-tests for the differences of inter-activity time length longer than one hour between different groups of developers in *Case I* and *Case II*.

T-test	$k \leq 50$	$k > 50$	T-value	Significance
Case I	181.4 (h)	83.0 (h)	20.5	$p < 10^{-6}$
Case II	198.0 (h)	87.2 (h)	22.0	$p < 10^{-6}$

committing rhythms of developers, although these two characters are strongly correlated with each other. This phenomenon suggests that the developments of these OSS projects may be more likely determined by the strong responsibility of the developers.

TABLE 4.4. T-tests for the differences between simulated and real evaluation and response latencies.

T-test	simulated	Real	T-value	Significance
Evaluation	89.9 (h)	58.5 (h)	8.21	$p < 10^{-6}$
Response	97.0 (h)	57.2 (h)	9.92	$p < 10^{-6}$

4.4.2. Communication and committing activities accelerate each other . Here, we use a time-series based method. The box-and-whisker diagrams for real and simulated evaluation latencies and response latencies of all the developers are shown in Fig. 4.7 (a) and (c), respectively. Since both evaluation and response latencies are highly skewed, *i.e.*, they may have some extremely large values, the boxes are flattened in these two figures. In order to present the differences of evaluation and response latencies between real and simulated cases more clearly, the y-axes are log-transformed and the corresponding results are shown in Fig. 4.7 (b) and (d), respectively, where one can see that both the real evaluation and response latencies are shorter than simulated ones. This phenomenon seems to suggest that these two kinds of activities may accelerate each other in reality. However, this result can be claimed only when the differences between the real evaluation and response latencies and the simulated ones are statistically significant.

The statistics of the T-tests are shown in TABLE 4.4. It is found that the average real response and evaluation latencies equal to 58.5 (h) and 57.2 (h), which are much shorter than the average simulated evaluation and response latencies that equal to 89.9 (h) and 97.0, respectively. The T-tests show that the differences are significant with relatively large T-values 8.21 and 9.92. According to this result and that obtained by the network based method, it is reasonable to say that, statistically, the communication activities can accelerate committing activities in reality. Note that, recent studies on human activities suggest that real distributions of inter-activity time may have relatively heavy tails [**94–96**], *i.e.*, there may be activities separated by long periods of inactivity, which may result in more extremely long evaluation or response latencies in the real situation than in the simulated situation, as shown in Fig. 4.7 (a). However, since the numbers of these extremely long latencies are very small, they will hardly influence the results presented here.

4.5. Conclusion and Discussion

In this paper, the network and time-series based methods are proposed to quantify the influence of social communications on working rhythms by analyzing the communication and committing data of 31 OSS projects in about 10 years, where some new definitions, such as evaluation and response latencies, and a mechanism to generate simulated communication time-series are introduced. Based on these methods, it is found that the developers with higher social status always have relatively shorter average inter-activity time and the average real evaluation and response latencies are also shorter than the average simulated ones, which suggests that social communications may accelerate committing rhythms of developers in reality. These findings can help researchers better understand the evolution mechanism of OSS systems, and then further help to design more efficient software engineering groups.

In the future, this work can be further expanded in the following several ways. First, the network and time-series methods can be used together in order to reveal whether the individuals response at different rhythms for the evaluations from others of different social status. This issue is important because it involves the efficiency and fairness and thus may determine the success of focused systems to a certain extent. Second, other microscopic multi-activities patterns need to be revealed, because more patterns will definitely provide more information about the interaction between communication and committing activities. Finally, based on these metrics, the co-evolution between different activities can be modeled.

CHAPTER 5

Tracing Distributed Collaborative Development in Apache Software Foundation Projects

5.1. Introduction

Teamwork's advantages are enjoyed in many walks of life, by humans [**97**] and other animals [**98**]. And those advantages become even more apparent in the digital realm, where people don't even have to be geographically near. Teams can be pre-meditated and serve a concrete purpose, *e.g.,* two people carrying a couch up the stairs, or group of undergrads building Facebook; then the outcome clearly benefits from the team's existence. Not surprisingly, team forming is taxing; those who team up incur costs related to the overhead of finding partners and coordinating with them. The larger the teams the higher this overhead [**99**, **100**]. In fact, managing large teams effectively is a top reason for the invention of organizational structure, and the subject of much managerial science research [**101**, **102**].

But in many occasions, teams also grow organically, without much explicit organization at any given growth step, thus reducing costs related to teaming up [**103**]. Examples include ants and bees swarming, schools of fish changing direction on a dime [**104**], and Open Source Software teams. In many ways, developing Open Source Software (OSS) code depends on teamwork, as many developers work toward the same goal [**105**]. Often teams in OSS form organically, as needed, though persistent organizational structures exist in some of them. [1] Teaming up in OSS serves many purposes, including finishing tasks which require many person-hours, and tasks which require varied expertise. It is also often implicit. Large OSS are known to exhibit latent communities which follow a hierarchical structure [**106**], organized around the socio-technical activities of developers. But how prevalent are those implicit, self-organized teams? Are

[1]Linus Torvalds runs the Linux project in a more centralized fashion, depending on his lieutenants for decisions regarding which new code filters up to him.

they persistent? Do some people have higher preference for working collaboratively because of their work styles? And does working on a team impart any beneficial effects on the team members in terms of efficacy?

Leveraging the availability of trace data from publicly available OSS projects, we have recently studied in 6 OSS projects the benefits of OSS developer collaborations by tracing pairwise synchronous code development, i.e., when two developers work on the same file around the same time [**107**]. In that setting, we found that two-person teams exist and are strongly affecting developer productivity. There, we left open the cases of synchronous code development in sets of sizes 3 and more people, due to lacking the technology to discover larger teams. The difficulty in scaling up the study from teams of size 2 to larger teams lies in the fact that it is non-trivial to implicate people in teamwork when dealing with OSS projects. This is true for many obvious reasons such as lack of central management and users having free will to work with or on whatever strikes their fancy at any given moment. However, techniques arising from self-organizational studies and research in task synchronization between actors [**107**] help in setting the frameworks for analyzing heterogeneous trace data abundant in empirical software engineering.

Here, we are interested in how collaborative distributed development scales beyond two people, to teams of three and more developers, when we operationalize it as synchronous distributed development. Our goal is to infer, or trace, putative teams from the large-scale data traces of developers' technical activities in Apache Software Foundation (ASF) OSS projects. We chose ASF for several reasons including ASF's multiple active and popular projects, and potential synergy with an already existing body of research [**107–113**]. We build on prior work on latent communities [**106**] and code development collaboration [**107**]. From the latter we take a definition of distributed collaboration and extend it to larger groups, in a way which easily lends itself to use with trace data of developer activities.

Like in our precursor work [**107**], we narrow down the definition of collaboration so that we can practically detect them from trace longitudinal data of commits. To consider a group of developers as having a collaboration, their technical actions, *i.e.,* code contributions, should be able to affect those of others. Physical and temporal proximity of code snippets being worked on by different developers increase the chance of merge conflicts thereby making it more likely for one developer to affect the other's contribution. We leverage this intuition to identify putative distributed collaborative groups in this study.

We call a putative *Collaborative Group (CoG)* a set of developers who can be identified via their code commits as working in close *code proximity* and *temporal proximity* to each other.

We chose our temporal and code proximity to be *one week* and a *package*, respectively. We will elaborate on these choices later in Section 5.4.3 extensively. With these parameters, our practical definition of a CoG becomes: "A set of developers working on the same set of packages within a time interval of 1 week from each other." From this point on, we interchangeably use collaboration, collaborative development, and distributed collaborative development, to refer to the above definition.

In the rest of the paper we describe how we used the above definition on longitudinal trace data from 26 ASF OSS projects to do the following.

- We develop a robust method for tracing putative CoGs from trace OSS data and use it to characterize distributed collaboration levels in 26 ASF projects;
- We measure developers' group collaboration levels, with respect to the time, commits, and lines of code contributed, finding that not all people spend a significant amount of time collaborating with others, but in most cases, when they do collaborate, their commits per unit time increases;
- We find that developers with higher focus on specific packages are also less likely to collaborate, while packages with higher ownership, as expected, are less likely the subject of distributed collaboration;
- We find that in terms of LOC added and deleted per commit, individual effort decreases during collaboration for 17 projects, while code growth decreases for 10 and increases for 2 projects, suggesting that while collaborating, developers arguably contend with problems of some increased complexity; and
- We ran a survey of ASF developers which provides input to our modeling assumptions, and anecdotal evidence for our results.

We describe the theory and our research questions in Section 2, followed by related works in Section 3. The methodology, results, and conclusions, are in Sections 4, 5, and 6, respectively.

5.2. Theory and Research Questions

Our first goal is to find putative CoGs and quantify their prevalence in our sample of ASF OSS projects. We do that by developing an algorithm for reasonably capturing the above definition of putative CoGs. In addition to their counts, we also want to know how their prevalence stacks against a random model of contributions. Since OSS projects also make publicly available some communication records between developers, validation of theses putative CoGs is plausible.

Research Question 5: How prevalent are putative CoGs, of size 3 and more? Do they occur more frequently than by chance? And is there evidence in the trace data that they are not just quantitative artifacts but real teams?

Having a way to count CoGs, we next turn to understanding programmer's participation in those groups, and if their preference for participation changes over time. In other words, we want to understand how developers' split their contributions between periods of collaboration and solitary work. This would show how prevalent co-development is from a personal perspective.

Research Question 6: How frequently do developers work in groups? Does preference for working in groups vs working solo change over time?

Next we turn to questions having to do with the effects of group work on the developer performance and technical output. The amount of one's focus on a particular file vs. distributing their activities equally across multiple files is an important contribution measure for developers. It can be quantified using a developer's contribution entropy. On the other hand, code ownership refers to the flip side, the entropy of contributions to each file, by developers. It has been shown that focused developers are less likely to introduce defects [**114**], and Windows binaries with higher ownership may include fewer defects [**115**]. It is reasonable to think that while in a group, a developer's focus may decrease and thus result in higher defects and lower productivity. So, in the next part we are interested to see how co-development correlates with developer focus and file ownership.

Research Question 7: Does distributed collaborative development correlate with developer focus? *i.e.,* do more focused developers participate in fewer CoGs?How about package ownership, do packages that are subject of co-development display a particularly high or low code ownership?

And finally we wish to see how distributed collaboration associates with individual productivity. Two straightforward metrics of productivity [**107**] are the code effort and code growth per developer for each file, defined as (1) code effort: LOC added plus LOC deleted per commit. (2) code growth: LOC added minus LOC deleted per commit. The code effort per file describes the amount of work performed on each file at each commit, while the code growth per file explains how much the size of a file grows per commit. These metrics can tell us whether developers are working more or less per file during distributed collaborative development.

Research Question 8: Do developers experience a notable change in effort or code growth during co-development? In particular, is there a pattern to the change as suggested by the data, *i.e.,* in the same direction, over all projects and developers?

5.3. Related work

Our work can be broadly contextualized into research on *Collaborative Software Engineering*, where the emphasis is on the mechanisms and outcomes of collaboration [**116**]. Prior work has addressed various challenges in this area [**117**], such as how to leverage others' experience, and what effective collaboration entails. Avritzer and Paulish provide a comparison of common processes utilized in multi-site software development [**118**]. Scacchi summarized the empirical studies of OSS projects, and describes collaboration practices at individual, project and cross-project level [**119**] and Lin has elaborated on the dynamics of collaboration between the Free/Libre OSS (FLOSS) community and for-profit corporations [**120**].

Tools and platforms for collaborating in software development have been described as enabling technologies [**121**], a key component in reducing the potential harmful effects of distance collaboration in the so called *Global Software Engineering* where the development teams are geographically distributed [**122**]. Holmstrom *et al.* use a case study to identify and highlight the key challenges in global software development, specially in regards to temporal, geographical and socio-cultural distances [**123**]. Sarma *et al.* outlined the main challenges organizations face in achieving *congruence* [**124**]. Grechanik *et al.* study the shortcomings of outsourced quality assurance teams and argue that new approaches are needed to overcome these issues [**125**]. Herbsleb *et al.* measure the costs of cross-site development vs same-site work and find out that cross-site work takes longer and requires more people to complete [**126**]. Al-Ani and Edwards studied the communication patterns of a fortune 500 company and found that the overhead for synchronous communication is unacceptably high [**127**]. They also show that the patterns of communication evolve from the initial inception to address the development needs. Jalali and Wohlin reviewed the literature regarding the application of Agile Methods in Global Software development [**128**].

Carmel presents some of the successful practices and techniques to make the best of distributed software development [**129**], which has also been the topic of several other papers [**130–132**]. Herbsleb outlines a desired vision of global development and expresses the need for a systematic understanding of the driving force of an effective coordination [**133**].

Interestingly, Takhteyev and Hilts look into the distribution of users in the GitHub ecosystem and find a strong local bias in contribution and attention [**134**]. This suggests that distributed development in many OSS is less about geographical distance and more about remote contributions and lack of physical presence. Nguyen *et al.* also showed that geographical distribution did not play a significant role in coordination response times [**135**].

Nakakoji *et al.* describe two different types of communication in software development: *coordination communication* and *expertise communication* [**136**]. They provide nine outlines for properly supporting expertise communication in OSS projects. Redmiles *et al.* introduce *Continuous Coordination* as a paradigm for coordination and communication that addresses some of the issues faced in global software engineering [**137**]. Later, Sarma *et al.* evaluated the continuous coordination tools available and their usefulness through an evaluation framework called DESMET [**138**].

In this context, our work stems from the fact that although there are many research papers on *Global Software Engineering*, there is still much to learn [**139**], and most of the research in this area are qualitative and there are very few empirical studies [**140**]. A number of these works are summarized by Šmite *et al.* [**141**]. Most directly related to this paper is a recent study by Xuan and Filkov, in which synchronous co-development was studied among pairs of developers in 6 OSS projects using trace data from commits and mailing list posts [**107**]. The co-development discovered was very strong, indicating that when collaborating together on the same set of files, developers communicate more frequently and *add* more lines of code than they *delete*. In this paper, we focus on all groups, and not just those of size 2. We aim to build upon this work by extending to groups of larger sizes and moving beyond files to packages as a notion of code proximity, which will allow us to identify collaborations that were previously undetected. This gives us a higher level view, where we will see groups which potentially form towards completion of specific tasks, rather than a set of pair-wise collaborations that are not connected to each other. We also expand our study to a wider range of projects, allowing us to provide a more complete insight on collaborative development.

Pinzger and Hall utilize a similar concept of code proximity to develop a collaboration and communication visualization tool for Eclipse [**142**]. Their tool helps identify collaborators on certain components, however, their approach is less concerned with a concrete definition of collaboration, and rather leaves its definition at the hands of the user through the tuning of the tool's parameters. Jermakovics *et al.* also use the notion of code proximity to identify the collaboration network in software projects [**143**]. Their approach however disregards any notion of temporal proximity which can lead to identifying unrealistic long-term collaborations. Caglayan *et al.* have also developed a community detection based approach to identifying teams [**144**] but their approach makes identified teams less suitable for the study of developers interactions. As mentioned, there are very few works on quantifying collaboration in OSS, most of which we have discussed here. Still there is a certain lack of solid quantitative methods to identify collaboration across OSS [**116**] which our work seeks to address.

With respect to social organization, Bird *et al.* have shown that developers organize themselves into groups and communities based on their social activities [**106**]. They have also shown that there is a socio-technical congruence between the software artifact modularity, and its developers' social groups. Panichella *et al.* extended this study by studying how these groups evolve and change over time [**145**]. Their study shows team structure is not stable and evolves over time, with respect to cohesiveness of files teams work on. Both of these studies are based on social networks extracted from the developer mailing lists; our current study is complementary, in that we look at groups arising from synchronous commit patterns. Robertsa *et al.* also used communication patterns and identified a strong group at the core of the Apache HTTP project [**146**]. Damian *et al.* studied communication and coordination patterns in an IBM team and found that task related social-network are ever-evolving and different from their initial conception [**147**]. Kakimoto *et al.* use Social Network Analysis to study the communication patterns in SourceForge projects and find a intensification of communications among members with different roles around the time of a release [**148**].

Hertel *et al.* studied the motivations of developers for joining specific subsystems [**19**] through a survey of 141 Linux Kernel developers. They observe that *valence*, *instrumentality* and *self-efficacy* are driving developers towards increased participation, patch submission and time spent in the project. However beneficial teaming up might be, Mockus showed organizational volatility increases the probability of software defects [**149**]. In terms of overhead incurred for teaming up, Adams *et al.* studied the Brooks' law effect on OSS [**150**]. They find that coordination costs increase only in a specific phase of a project and after that, it becomes "quasi-constant". Nagappan *et al.* define a set of metrics based on organizational structure and show that they are able to predict defect-proneness better than classic code metrics such as code churn [**151**]. While their metrics are certainly interesting, they are defined based on an established well defined organizational structure, and many of the metrics are not applicable to our implicit and non-hierarchical definition of CoGs.

Dabbish *et al.* interviewed a group of GitHub developers and found that transparency in large development efforts support efficiency and collaboration [**152**]. In a case-study, Herbsleb and Gringer [**153**] show that communication is essential to code development and that distribution of activities can hinder communication. Cataldo and Herbsleb [**154**] show that a gap between coordination requirements and actual coordination can increase software failures. Luther *et al.* identified frequent communication and high activity of members and leaders as a success factor in online collaborations, including OSS [**155**]. Nakakoji *et al.* describe OSS projects as participative system and hypothesize the that the change in participants' perceived values in a project may may characterize the evolution of that project and its community [**156**].

A topic that is highly tangential to collaboration is *code ownership*. Rahman and Devanbu have studied the effect of code ownership in OSS [**114**] and find that defects are more likely from contributions of a single developer. Bird *et al.* studied proprietary software *viz.* Windows 7 and Vista [**115**]. Both studies above are in agreement and report that files or binaries with many minor contributors are more likely to contain defects. Focault *et al.* attempted to replicate the results by Bird *et al.* [**115**] in OSS, but their results was in contrast to the previous works' findings [**157**]. However, the difference in their methodology and corpora studied, makes the comparison of their results a little controversial. *Developer focus* is another issue that can be seen as the dual of code ownership. Posnett *et al.* show that these two measures can be unified under a more generalized view [**158**]. We use those measures in this paper.

While, to our knowledge, there is no direct study of the correlation between group collaboration and performance and productivity in self-organized teams, in the field of managerial science the concept of teamwork and its correlation with performance has been touched on by Brooks [**159**], Kuipers and de Witte [**160**], Moe *et al.* [**161**], and others. They find that improved cooperation is needed to gain higher productivity, but team size does not figure linearly. They also show that increasing task delegation reduces defects in a team's product. These results are for centrally managed, well developed teams.

5.4. Methodology

Figure 5.1 provides an overview of the process of mining and parsing data, extracting CoGs and multiple verification steps, as described below.

5.4.1. Mining Source Code Repositories. We mined the git source code repositories for 26 ASF projects; summary is in Table 5.1. Data gathering on the mailing lists and the repositories was performed at

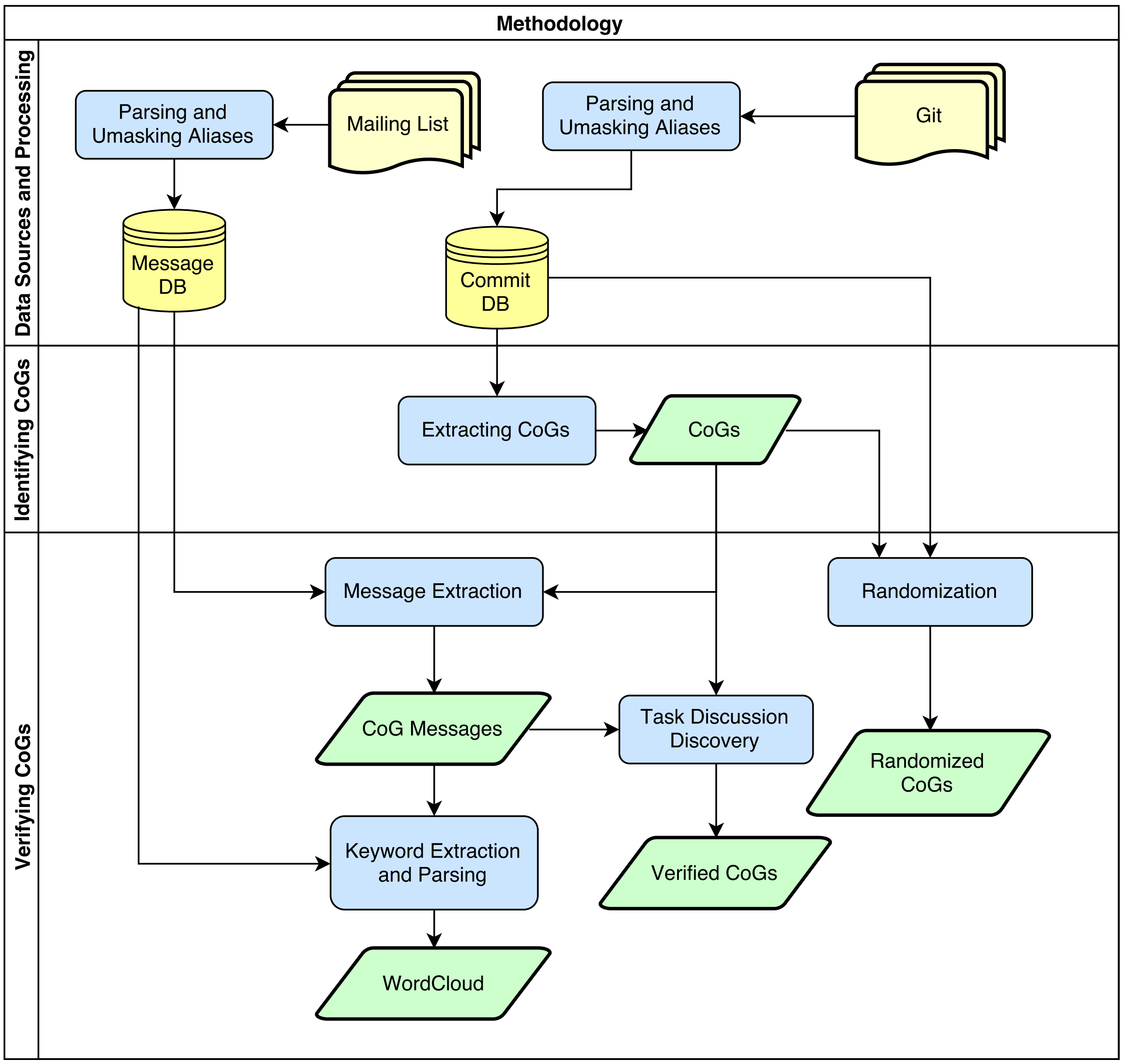


FIGURE 5.1. The process diagram of the data gathering, identification and verification steps as described in the paper.

different times and the times shown in Table 5.1 represents the intersection of both of these datasets' date range. All of the selected projects have at least 5 developers and 2 years worth of commit history that is maintained in their Git repository.

The `git log` command gives us a complete record of each commit, along with its id, date, and committer. We then parse the output of `git show` for each commit to compile a list of files that are changed

TABLE 5.1. List of OSS projects used in this study. The end date in most of the projects is the date that data collection was performed.

Projects	Devs	Start	End	Projects	Devs	Start	End
abdera	13	2006-06	2011-12	ivy	9	2005-06	2012-01
activemq	28	2005-12	2012-01	log4j	18	2000-11	2012-01
ant	44	2000-01	2012-02	log4net	7	2004-01	2011-12
avro	12	2009-04	2011-12	log4php	9	2004-01	2012-01
axis2_c	24	2005-09	2010-02	lucene	41	2001-09	2011-01
camel	31	2007-03	2012-01	mahout	15	2008-01	2012-01
cassandra	13	2009-03	2011-12	nutch	16	2005-01	2012-01
cayenne	20	2002-03	2012-01	ode	17	2006-05	2011-12
cxf	45	2005-07	2012-01	openejb	38	2002-01	2012-01
derby	35	2004-08	2012-01	pluto	24	2003-09	2011-09
hadoop_hdfs	25	2009-05	2011-06	solr	19	2006-01	2011-03
harmony	25	2005-09	2011-07	wicket	24	2004-09	2011-12
hive	18	2008-09	2012-01	xerces2_j	33	1999-11	2012-01

with that commit, and each entry contains the number of lines of code that has been added or deleted during that change.

We made a choice to only use source code files in our analysis, and identified them with the methodology of Geominne *et al.* [**162**]. Code is the main material and substance of OSS and also the main focus of developers, thus we study them as to extract collaborative development. Code files also make up the roughly 90% of files in our dataset, so limiting our study to this subset will not drastically change the number of identified CoGs, but it will improve their quality. For example, there are files that are touched every time a developer submits changes, *e.g.,* "changes.txt". These would produce large CoGs for very extended durations, which is meaningless and skews our results and so they must be removed.

5.4.2. Merging Aliases. Merging duplicate aliases is necessary for two reasons: (1) A person identified as two or more people might be working on a single file using a number of his/her aliases, and thus could be erroneously recognized as a group; (2) A person with multiple ids might be working on two or more different files using different aliases, and thus the attribution of focus will be wrong.

Due to the small number of developers in each project (in order of tens), the merging process was done manually. All names and email IDs were checked and those with similar name, or similar email or names that highly suggest similarity were merged *e.g.,* (John Smith, smith@gmail.com), (Smith, John@smith.com), (John S., J.smith@ucdavis.edu) are considered to be the same person.

5.4.3. Extracting Groups. We make the precise definition of a putative CoG by designing an algorithm which identifies sets of developers whose commits overlap in code and time. It depends on both *temporal proximity*, Δt, the maximum time separation between two consecutive CoG commits, and *code proximity*, which captures a level of modeling at which CoGs work. We reason about the appropriate practical choices for these later, after we present the generic algorithm.

(1) For each code proximity, we process in increasing temporal order the commits therein, and derive a list of *contribution sequences*. Each sequence corresponds to the contributions of a set of people to that code proximity.
(2) A commit is added to a sequence if it lies within a Δt time apart from the last one already in the sequence. If no prior commit lies within Δt of the current one being processed, we start a new contribution sequence with it.
(3) After all commits have been processed, we remove all final contribution sequences solely comprised of commits by a single developer.
(4) We also excise consecutive commits by the same person occurring at the beginning or the end. The reason is that the head or tail sequences usually represent a time of solitary development before new developers joined the existing one and formed a group. For example, the sequence

$$F_1 : P_{1,1}, P_{1,5}, P_{1,8}, P_{2,8}, P_{2,9}, P_{1,10}, P_{2,12}, P_{2,13}, P_{2,15}$$

is reduced to:

$$F_1 : P_{1,8}, P_{2,8}, P_{2,9}, P_{1,10}, P_{2,12}.$$

Here $F_i : \ldots$ shows the sequence of changes made to code proximity i, and $P_{j,k}$ denotes changes made by person j at time k.
(5) Since developers in the same CoG may be working on more than one code proximity at the same time, *e.g.,* they may have divided tasks among each other, we merge contribution sequences of the same developer set if their lifespans are within Δt of each other, even if they belong to different code proximities. For example, if we have

$$F_1 : P_{1,8}, P_{2,8}, P_{2,9}, P_{1,10}, P_{2,12}$$

$$F_2 : P_{2,10}, P_{1,11}, P_{2,11}, P_{2,14},$$

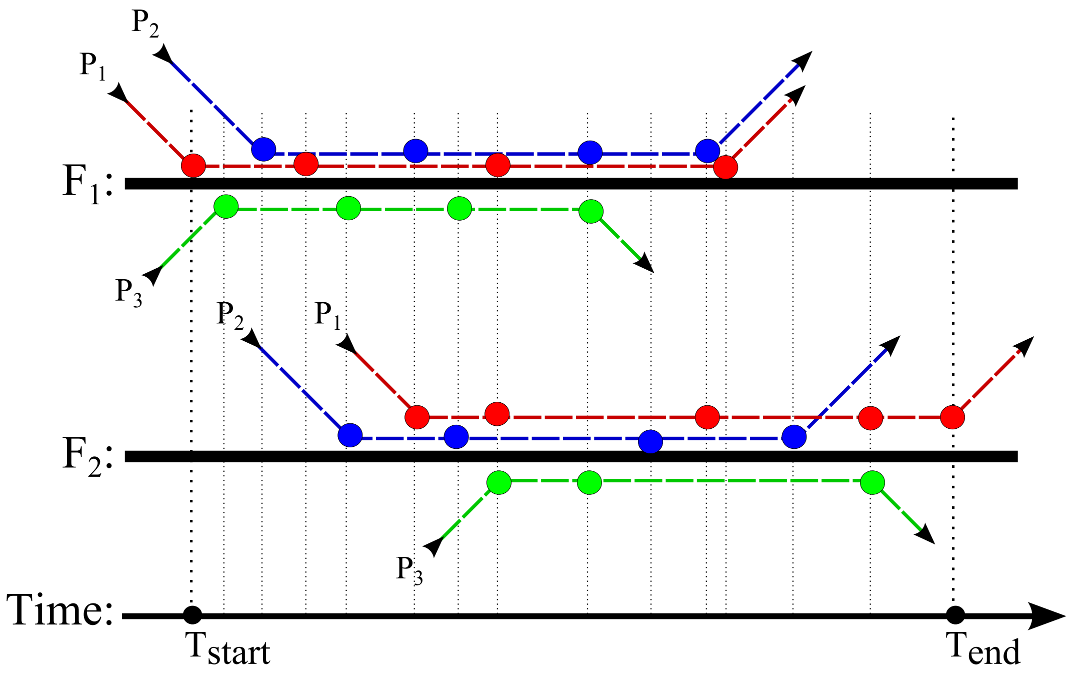


FIGURE 5.2. An illustration of a sample CoG of size 3 (developers P_1, P_2 and P_3), and strength 2 (code proximities F_1 and F_2). The lifespan of this group is $l = T_{end} - T_{start}$.

we merge them into:

$$F_{1,2} : P_{1,8}, P_{2,8}, P_{2,9}, P_{1,10}, P_{2,10}, P_{1,11}, P_{2,11}, P_{2,12}, P_{2,14}.$$

We extract the putative CoGs from those last sequences.

The number of developers in each putative CoG is its "size" and the number of modules in each group is its "strength". A sample putative CoG can be seen in Figure 5.2. This definition of a CoG is closely related to the concepts of "implicit teams" and "succession" in Mockus' research [**163**].

5.4.4. Choices for Temporal and Code Proximity. We thought of two possible choices for code proximity, a file and a package, and based on the existing literature [**142**,**143**], both seem to be valid choices. The most obvious choice for code proximity is that of a file, and this level of interaction is easy to mine directly from the commit logs. However, it may not provide the appropriate level of modeling putative CoGs as, often, adding a feature or fixing a bug requires touching multiple files in the same package, and even beyond. The consequence is that when developers divide tasks, they may split those files into non-overlapping groups

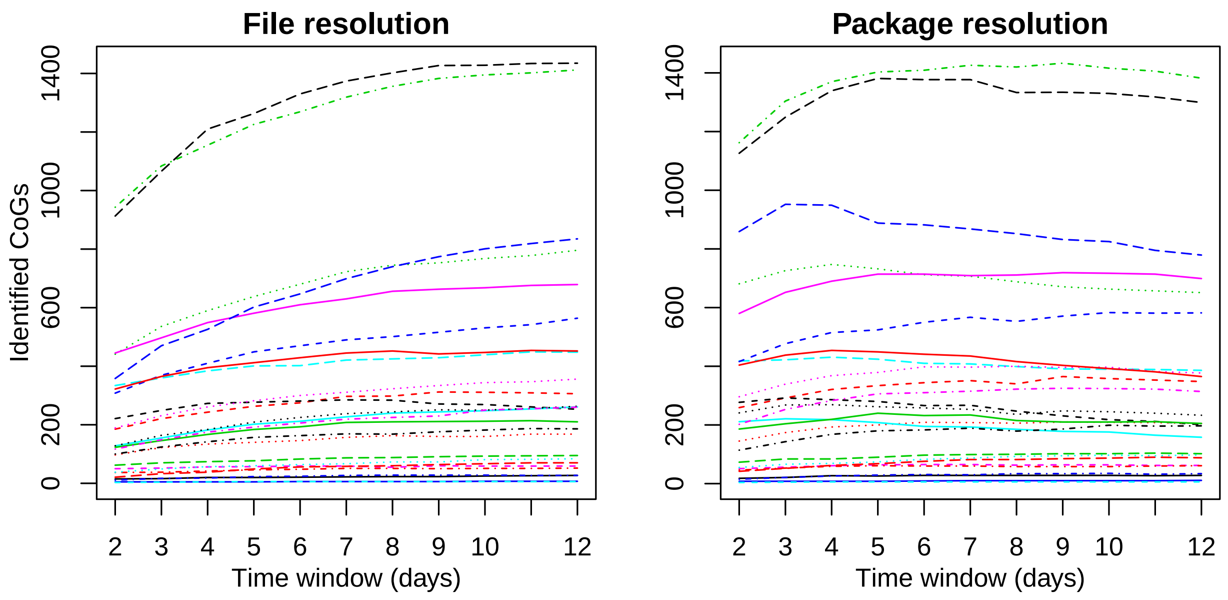


FIGURE 5.3. The effect of time windows size on number of groups identified in projects. Each line represents one of the 26 projects.

in order to maximize throughput among them. Thus, people touching different files in the same package may still be working together. For this reason, we decide to use package-level as our code proximity. [2]

To chose the appropriate Δt we reasoned that, on average, contribution to an ASF OSS project may not be a developer's top priority, so he may want to postpone it a day or two, or even do it on weekends. On the other hand, too big a time window would result in two non-related changes to a file to be counted towards as collaboration. We chose Δt to be one week (7 days) in this study.

To make sure our choices for temporal and code proximity were sensible, we applied the above algorithm with a range of possible parameter values: file or package level for code proximity, and between 2 to 12 days for Δt. The number of CoGs identified in each project for all combinations are plotted in Figure 5.3. Based on the stability of the results and the above reasoning we have concluded that our choices are appropriate. [3]

We also note that our choices are also greatly in agreement with the prior study, that used slightly finer time intervals [**107**].

[2]Most of our studied projects are written in Java where files within the same file directory are considered to be in the same package. The three non-java projects, "axis2_c", "log4net", and "log4php", use the same file structure as their Java counterparts,"axis2_java" and "log4j".

[3]We also generated all the results for Tables 5.3, 5.6 and 5.10 for choices of 2 and 5 days for Δt. While the results were slightly different, the overall theme of the tables remained consistent with our original choice.

5.4.5. Mining Developer Mailing Lists. Mailing lists serve as the main communication hubs for OSS projects [**68**, **164**], and have been widely studied in recent years [**109**, **111**]. To evaluate whether putative CoG members coordinate among themselves, we extract messages between developers from the developer mailing lists using existing in-house parsers. While there are a collection of mailing-list parsers available, the need for compatibility with our design choices and existing databases drove us into using our own tools. While we have not formally validated the correctness of our scripts, the fact that their artifacts were checked through multiple publications [**41**, **109**, **111**, **165**], gives us confidence in their results. One of the main challenges in mining these lists is unmasking user aliases. The large number of participants in the mailing lists makes it next to impossible to manually curate the participants' aliases. We employ a technique introduced by Bird *et al.* [**41**], which we later improved upon [**109**], to semi-automatically unmask aliases. We refrain from going into more detail on this issue as it has been discussed extensively before [**109**].[4]

Messages sent to a mailing list are broadcast to all subscribed participants. Person-to-person exchanges can be inferred, using the "in-reply-to" field in the messages. When person B sends a broadcast message in reply to another broadcast message originally sent by person A to the list, we infer that B intends to communicate with A [**41**], and record it as a message from B to A.

5.4.6. Extracting Message Text. To study the content of the messages posted by developers, we first process them to eliminate artifacts. Messages usually contain the body of the message they are in response to. To remove those we filter out patterns highlighting prior text, starting each line with a special character such as '>', '|', '}' , or '*', or all the lines after tags such as: "----begin forwarded text----". We also remove punctuation and numbers, and convert all messages to lowercase letters.

We use these texts to create "word clouds". When creating word clouds from text, it is common to perform *stemming*, and to remove *stop words*. We omitted stemming because we are interested in looking at occurrences of verbs and nouns, and stemming works against this distinction. We did remove stop words, but not the *standard* English stop words. We want to see the occurrence of words such as "you", "I", and "us". Apart from these common but interesting words, other very common words such as "the", "this", and "that" were removed.

[4]Since the methodology in this section is the focus of previous works [**41**, **109**], we did not include those scripts and tools in the supplementary materials. Interested readers can find them on the original papers supplementary materials at `http://www.gharehyazie.com/supplementary/oss/`.

TABLE 5.2. A sample changelog along with a valid and two invalid randomization examples. Sample 1 is invalid because file 3 is changed before its first change in the original data. Sample 2 is invalid because file 1 and file 3 are changed with a different frequency than original data.

Original		Valid		Invalid 1		Invalid 2	
logid	fileid	logid	fileid	logid	fileid	logid	fileid
1	1	1	1	1	1	1	2
1	2	2	2	$\boxed{1}$	$\boxed{3}$	2	$\boxed{3}$
2	1	3	1	2	2	3	4
2	3	3	2	3	1	3	$\boxed{1}$
3	4	3	4	3	4	4	$\boxed{1}$
3	1	4	1	4	2	4	2
4	1	4	4	4	1	5	$\boxed{1}$
5	2	5	1	5	1	5	$\boxed{3}$
5	4	5	3	5	4	5	2

5.4.7. Generating Randomized Datasets. To evaluate the significance of the putative CoGs frequencies in each project, we compare their counts against a randomized baseline model, *i.e.,* chance collaboration. We generate a baseline distribution by randomizing the commit dataset for each project 20 separate times. This small number of random samples was imposed by the computational complexity of the operation and the total number of different projects; it still provides sufficient statistical power for our purposes.

We randomize the commit data by changing the files that a developer touches during a commit to a random set of files of equal cardinality, chosen from all available files at the time of that commit. This ensures the temporal order of the commits is preserved while the files changed during each commit are randomized. [5] We take care that files introduced later in a project are not changed before they were introduced in the project by only assigning a file to a commit after it had been introduced in the original data.

We also take care to keep the file size the same as in the empirical data. When we randomize files through commits, we preserve the number of times a file has been changed plus the number of lines added and deleted, thus file size remains similar to the original data, with respect to each time they are changed, *e.g.,* a file that has been changed three times has the same size after each commit in the randomized dataset that it has in the original dataset after the same number of commits, but *when* these changes were made may differ between the two. A sample randomization is presented in Table 5.2 for further clarification.

[5] The reason that we speak of files instead of packages at this stage is that commit datasets record files, and to randomize them, we have to randomize at a file level. All results extracted from these randomized datasets are still based on package level code proximity.

The alternative randomizations of a) shuffling people around, and b) shuffling the commit dates are not as desirable. The former randomization is not realistic, because each developer stays with a project for a limited time (around $2 - 3$ years). If we randomize developers across commits, it would appear that all developers are spread out thinner, in terms of commit numbers, but across the whole life span of the project (from 6 to 12+ years). The latter option is even worse. Not only it has the shortcomings of the previous alternative, but also distributes each file across the whole project time line, and it removes spikes in temporal commit activity on which the definition of group collaboration relies on.

5.4.8. Developer Survey. We designed a short survey to inform our assumptions and qualitatively verify our findings on the issue of collaborative development in ASF projects. The questions were designed to be concise and as unambiguous as possible. A copy of the questionnaire can be found in Appendix A

When selecting subjects for this survey, and designing the questions, we faced a number of challenges. The first one was that we needed developers who had participated in a project long enough to potentially have a meaningful form o collaboration with others. For example just one month of contributions is not long enough to form a solid opinion on the state of collaboration in a project, so we selected the developers with at least 2 years of contribution experience.

The second challenge was passage of time and memory recollection. As previously mentioned, the selected projects were at least two years old at the time of data gathering (March 2012). Many developers have since left the projects, and/or may not have a clear recollection of how things were, say 10 years ago. This affects us in two major ways. First, We cannot use the survey to directly verify our results through developer confirmation. For example, we cannot ask: *“were you collaborating with person X around date Y on module Z?”* This is too specific of a question and even if people do provide an answer, we cannot be very confident in their recollection of the events. We need to design the questions towards the general act of collaboration and cooperation than any specific instance. Second, we want to ask people with the most recent experience so that their recollection will be as accurate as possible, so we chose developers who were active at the end of the data gathering.

This leaves us with 85 developers, who we contacted through email and invited them to participate in the survey. Participation was voluntary and confidential, and was expected to take less than 10 minutes. We received 9 responses to the survey, a response rate of roughly 11%. Apart from the mentioned difficulties, several factors also contribute to the low response rate, mainly the fact that the questionnaire is classified as "junk mail" by many developers. As one of the corresponding developers mentioned: "Being an active OSS developer means I get asked to fill in surveys several times a month...What is the benefit for (project name), the ASF or myself?". Taking this into account rather explains our lower than expected response rate [**166**]. The lesson we took from this experience was that incentivization is crucial for high survey response rates.

5.4.9. Measures of Focus and Ownership. To reason about the relationship between the distribution of commits across packages and group collaboration, we use the *Developer Attention Focus* (DAF) and *Module Activity Focus* (MAF) measures [**158**]. These metrics were initially introduced by Bluthgen *et al.* [**167**] and here we give a brief description of them for completeness.

Let $W_{n,m} = \{w_{i,j}\}$ denote a commit matrix for a project of m developers and n modules, where $w_{i,j}$ denotes the number of commits by developer j to modules i. Then the total commits by developer j is $D_j = \sum_{i=1}^{n} w_{i,j}$ and the total commits that a module i receives is $M_i = \sum_{j=1}^{m} w_{i,j}$. Here *module* can be a file or a package. We use the latter in our measurements to be consistent with our definition of collaboration. The total commits to all modules is then $A = \sum_{i=1}^{n}\sum_{j=1}^{m} w_{i,j}$, the proportion of commits to module i is $r_i = M_i/A$, and the proportion of commits by developer j is $q_j = D_j/A$. The proportion of commits by developer j to module i is $q'_{i,j} = w_{i,j}/D_j$ and the proportion of commits to module i is $r'_{i,j} = w_{i,j}/M_i$.

Based on these definitions, the DAF and MAF metrics are given by:

$$\mathcal{DAF}_j = (\delta_j - \delta_{j_{min}})/(\delta_{j_{max}} - \delta_{j_{min}})$$

, and

$$\mathcal{MAF}_i = (\delta_i - \delta_{i_{min}})/(\delta_{i_{max}} - \delta_{i_{min}})$$

, where [**158**]:

$$\delta_j = \sum_{i=1}^{n}\left(q'_{ij}\ln\frac{q'_{ij}}{r_i}\right)$$

, and

$$\delta_i = \sum_{j=1}^{m}\left(r'_{ij}\ln\frac{r'_{ij}}{q_j}\right).$$

Based on these definitions, DAF and MAF range from 0 to 1. A low DAF (closer to 0) means that a developer has spread his commits across many files rather evenly, and a high DAF (closer to 1) means that all of their commits concern very few files. Similarly a low MAF would mean, that commits on a file come from many users, while a high MAF means most commits on that file are coming very few developers.

These measures of focus are based on cross entropy and can also be seen as related to entropy based inequality indices such as the Theil index used to aggregate software metrics [**168**, **169**].

5.5. Results and Discussion

TABLE 5.3. The number of identified and confirmed CoGs in each project, along with minimum, average and maximum number of CoGs in the randomized dataset.

	Collaborative Groups		Randomized Data		
	Identified	Confirmed	Min	Average	Max
abdera	28	14	35	41.5	48
activemq	351	161	1492	1544.0	1589
ant	707	299	2296	2335.8	2399
avro	30	24	112	127.4	141
axis2_c	408	129	780	813.8	841
camel	709	325	1952	2015.8	2066
cassandra	267	32	308	327.2	344
cayenne	208	43	356	380.9	406
cxf	1426	291	6980	7054.1	7126
derby	868	521	2101	2166.7	2226
hadoop_hdfs	193	35	316	340.0	362
harmony	65	17	125	133.2	143
hive	253	58	394	411.9	432
ivy	60	11	77	83.7	89
log4j	99	28	226	236.4	256
log4net	10	4	16	17.8	20
log4php	6	5	13	15.3	18
lucene	397	223	1105	1140.0	1171
mahout	189	111	535	556.8	577
nutch	82	41	195	203.2	214
ode	234	99	500	523.1	541
openejb	567	132	3327	3369.2	3413
pluto	87	45	223	230.5	239
solr	315	159	849	865.5	907
wicket	1377	220	2792	2840.1	2892
xerces2_j	435	20	1041	1076.1	1109

5.5.1. RQ1: Identification, Validation and Verification of CoGs. Using the algorithm above we identified putative CoGs in each of the 26 OSS projects described above. Table 5.3 gives the number of CoGs in each project along with the average, min and max number of CoGs in the randomized dataset. We also generate a normal distribution from the randomized datasets' number of identified CoGs and calculate the probability that the empirical results belong to the same distribution *i.e.,* the probability of the number of identified CoGs, given the normal distribution. The probability in all cases is smaller than 1%, making it

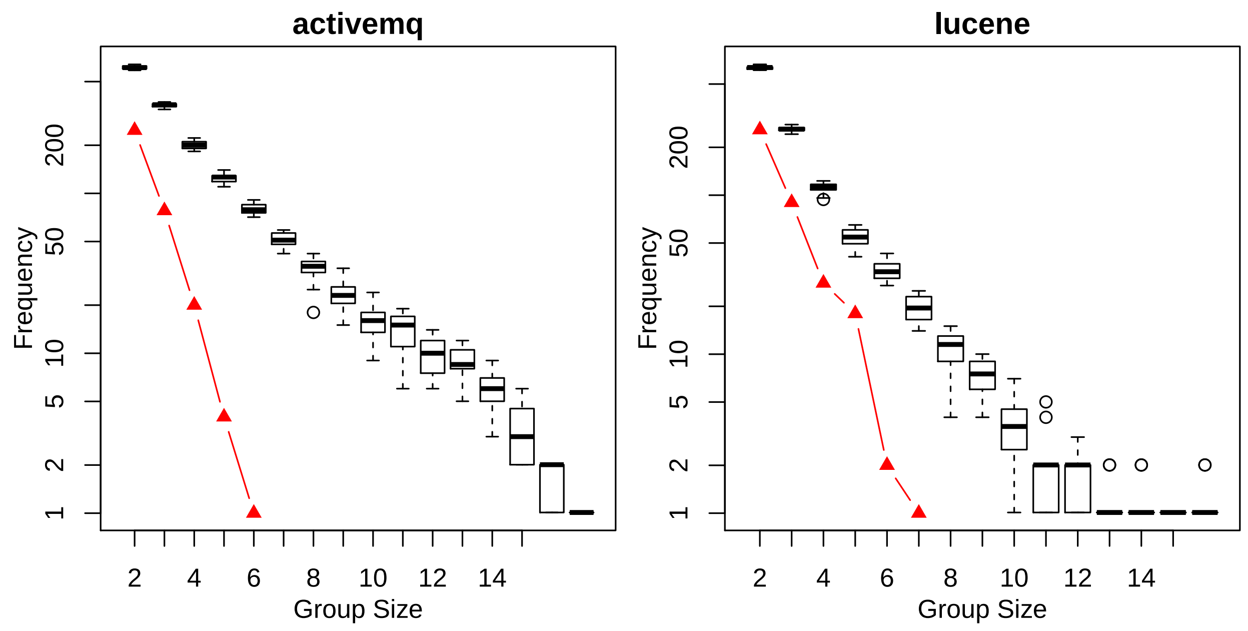


FIGURE 5.4. Distribution of CoGs' size in two OSS projects. The box-plots show the distribution of the randomized baseline models and the red lines shows the empirical results.

unlikely that our results are due to chance. The randomized models are meant not as a model of an actual developer collaboration, but rather as a way of excluding the most obvious random behavior.

We illustrate next the distribution of group sizes within and between projects. For two randomly selected projects, we have provided box-plots of the distribution of group sizes in our baseline random models in Figure 5.4. The red line shows the distribution of group sizes in our empirical data. The empirical data seems to follow the same distribution, but at significantly lower levels.Groups above a certain size do not occur in our empirical results.

5.5.1.1. *File Name Mentions as Evidence in Support of CoGs as Teams.* To further assess whether the extracted putative CoGs are meaningful, we look at the content of messages exchanged between CoG members.[6] We mined the *developer mailing list* archives for messages between developers, as described. For each putative CoG, we searched in all messages sent between developers for the names of files from their contribution sequences. [7] A putative CoG is considered *confirmed* if the exact name of at least one of those files is found in the subject or body of the message, during that CoG's lifespan. The results from

[6]We scanned by hand a number of CoGs and were able to identify via the contents of their messages that *developers were truly coordinating their collaboration* as predicted. That encouraged us to come up with the automated, but necessarily more simplistic, large-scale analysis, presented here.

[7]We search for files within the packages subject to collaboration since in technical discussions, file names occur naturally and more frequently than package names.

TABLE 5.4. The number of identified and confirmed CoGs in each project. This is similar to Table 5.3 but only done with CoGs extracted with $\Delta t = 2$ and $\Delta t = 5$.

	Collaborative Groups $\Delta t = 2$		Collaborative Groups $\Delta t = 5$	
	Identified	Confirmed	Identified	Confirmed
abdera	18	8	26	14
activemq	259	80	334	137
ant	681	217	732	284
avro	15	8	29	20
axis2_c	418	83	424	111
camel	580	266	714	333
cassandra	277	13	280	25
cayenne	145	21	200	38
cxf	1162	149	1403	252
derby	859	444	888	519
hadoop_hdfs	211	34	208	40
harmony	51	15	64	17
hive	241	46	262	59
ivy	46	5	61	10
log4j	73	15	90	26
log4net	8	4	8	4
log4php	4	4	6	5
lucene	296	161	379	203
mahout	114	53	180	94
nutch	42	20	68	35
ode	186	39	240	79
openejb	416	64	524	102
pluto	55	22	74	37
solr	202	92	306	143
wicket	1126	139	1381	195
xerces2_j	404	12	449	18

our approach are presented in Table 5.3, in the confirmed column. We managed to identify at least one of the files a putative CoG was inferred to have been working on in 38% of CoGs in all projects. This number was (not unexpectedly) much lower, around 16% on average, for projects where communications level was much lower than development activity *i.e.,* there were fewer messages in their mailing lists than commits in the repository. Interestingly, when we examine teams identified with a smaller *temporal proximity*, the percentage of confirmed teams drop to 30% for $\Delta t = 5$ and 26% for $\Delta t = 2$ Table 5.4. This further strengthens our confidence in our choice of *temporal proximity* for CoGs.

This approach likely underestimates the number of CoGs that coordinate via the mailing lists as it is possible that even though group members are communicating, file names may not be mentioned, or may be mentioned in an inexact form. It is also likely that they use other communication methods, as pointed out

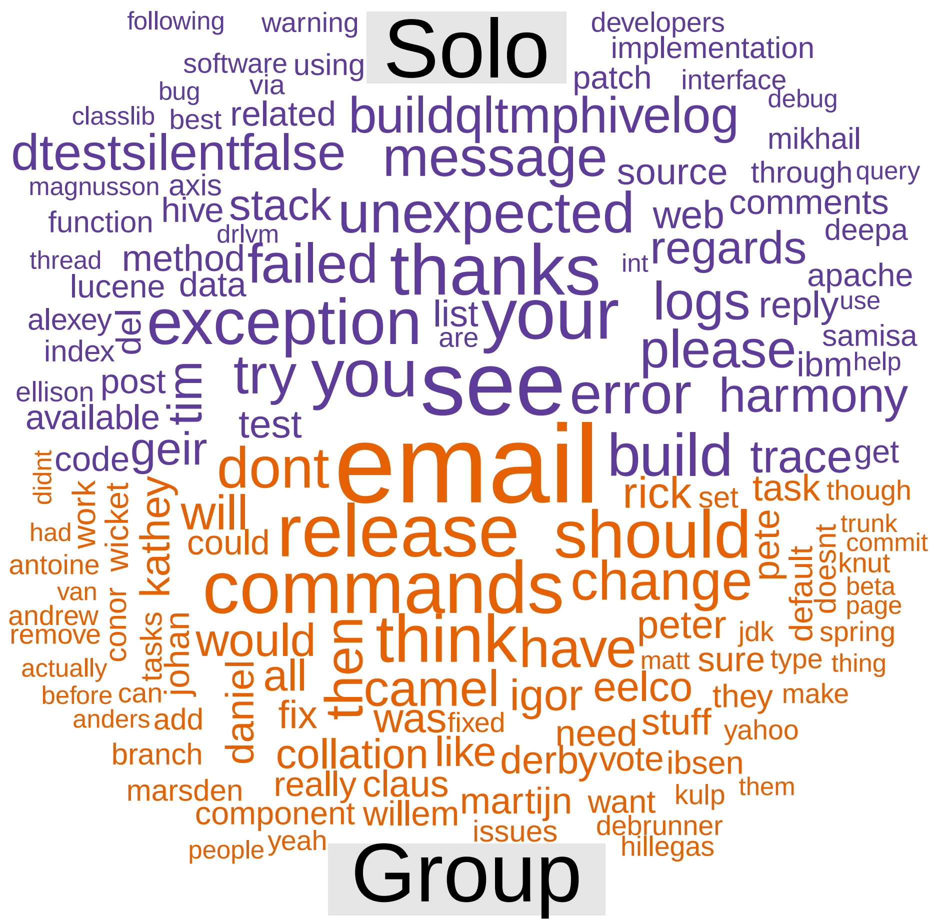


FIGURE 5.5. Difference word cloud of words used within messages within and outside of CoGs. The size of each word represents the extra prevalence it has in the corresponding group.

by developers in our survey, *e.g.,* IRC or personal email for coordination, even though it is discouraged by ASF.

Taking the underestimation into account, the confirmed numbers are strong evidence supportive of the thesis that a high number of CoGs coordinate and collaborate explicitly.

5.5.1.2. *Language Differences in Group vs. Solo Mode Communication.* We were also interested to learn whether there is any semantic difference between those messages posted while developers were involved in CoGs and while they were working solo. As previously mentioned, we extracted the text of each message and parsed their words. From them, we created two distributions of word frequencies: "group" word distribution, of all words used in messages posted by people during their time spent in CoGs, and

TABLE 5.5. Distribution of responses to agreement questions in the survey.

Question	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Working on the project was a collaborative effort	1	0	0	2	6
You actively attempted to "team up" with others to complete tasks	0	1	2	2	4
Collaboration increases productivity	0	0	1	3	5
Collaboration requires extra coordination and communication	1	1	1	4	2

"solo" word distribution, of all words used in messages posted by people during their time spent not in CoGs. Each distribution was normalized by the number of messages in each group. A comparison contrasting word cloud of the resulting word distributions is shown in Figure 5.5, where we see a stark difference between group-mode language and solo-mode language. [8]

We observe that while in group-mode, developers use more action words, such as "should", "change", "think", and "release", along with task oriented terms like "task", "work", "add" and "remove". These words are in great agreement with commit message language found by Maalej and Happel [**170**,**171**], which together suggest that CoG conversations may be more task oriented than solo developer messages.

In contrast, solo-mode messages are more ripe with words like "bug", "debug", "build", "trace" and "exception", hinting at greater focus on removing defects and improving quality rather than adding new features to the software. We also point out the prevalence of negative terms such as "don't" and "doesn't" in group messages versus solo-mode messages.

5.5.1.3. *Survey Results.* The survey of the ASF developers offers a valuable verification of our results. The developers who responded largely agreed with each other on most topics. Here we discuss the findings from our survey as they pertain to this RQ; others will be presented with the subsequent RQs. According to our survey results presented in Table 5.5:

Working on a project is a *collaborative* effort: 8 out of 9 Participants agreed or strongly agreed that OSS development is a collaborative effort rather than several solitary contributions. The one person who *strongly disagreed* with this, responded *"I tended to find that only implementing new features was collaborative..."* when asked *which of their tasks were more collaborative*. **Developers *choose* what to work on:** When we asked developers about how they choose and prioritize their tasks and if they choose what they work

[8]The word cloud was created using the "comparison.wordcloud" function in the "wordcloud" package in R.

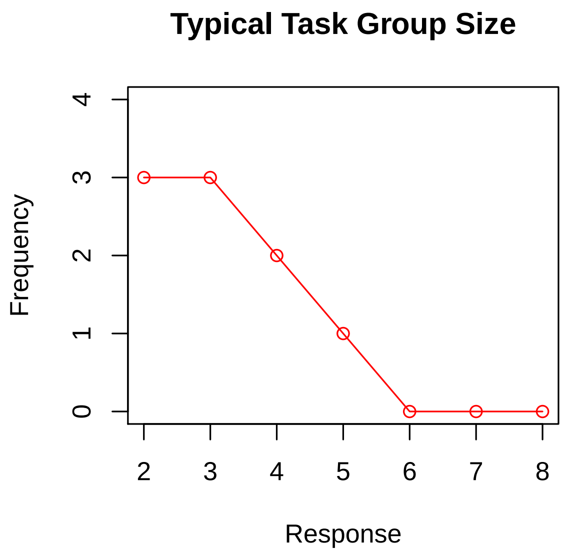


FIGURE 5.6. The frequency of responses to the typical number of collaborators on a task.

on, we got responses such as *"Choose my own tasks, whatever itches I want to scratch"*, *"I've usually selected tasks with which I was comfortable implementing them..."*, *"what ever needs to be done"*, *"...each worked in the area they were most comfortable with and also addressed issues that were critical..."*, and *"...I jump on any issue that affects reliability or correctness."*, all indicating that it was the developer's decision to work on a specific topic and region of the project, even if this decision was based on certain criteria and priorities. Thus, working on the same code proximity is by choice, not forced. **Developers *actively* team-up to complete tasks:** 6 out of 9 agreed or strongly agreed while 1 person disagreed with this, which implies that collaboration is explicit and what we find is not just an artificial artifact of our methodology.

We asked developers about the typical time to complete a task and most of them (6 out of 9) responded: **Completing typical tasks takes** $3 - 5$ **days**, and two others said $1 - 2$ days. The median lifespan of a CoG across all 26 projects is 6 days. This shows that identified CoGs more or less correspond to actual development tasks. Figure 5.6 also shows the response to **the typical number of people involved in a task ranges between** 2 **and** 5 with a downwards trend towards larger teams that highly matches the range of identified CoGs size as previously discussed Figure 5.4.

TABLE 5.6. Amount of person-days and commits in each project, separated by group vs. solitary development. Bold items in columns 3 and 6 represent values greater than 0.7 (significant values). Projects highlighted in the last column express greater commits per day during group vs. solitary development.

	Person Days (D)			Commits (C)			C/D	
	CoG	Solo	G/S	CoG	Solo	G/S	CoG	Solo
abdera	292	3912	0.07	358	1279	0.28	1.23	0.33
activemq	4670	17397	0.27	2855	4267	0.67	0.61	0.25
ant	13754	30986	0.44	6898	10532	0.65	0.50	0.34
avro	320	3647	0.09	169	973	0.17	0.53	0.27
axis2_c	5091	8210	0.62	4453	3372	**1.32**	0.87	0.41
camel	8955	12181	**0.74**	7008	8704	**0.81**	0.78	0.71
cassandra	4342	467	**9.30**	4155	4157	**1.00**	0.96	*8.90**
cayenne	2504	15565	0.16	2442	6241	0.39	0.98	0.40
cxf	10936	22971	0.48	7846	11265	**0.70**	0.72	0.49
derby	15227	13717	**1.11**	4979	5667	**0.88**	0.33	*0.41**
hadoop_hdfs	4339	5322	**0.82**	1068	1486	**0.72**	0.25	*0.28**
harmony	520	4976	0.10	437	1025	0.43	0.84	0.21
hive	4834	2067	**2.34**	1175	1559	**0.75**	0.24	*0.75**
ivy	876	5173	0.17	581	2141	0.27	0.66	0.41
log4j	1414	17639	0.08	1044	2752	0.38	0.74	0.16
log4net	66	4937	0.01	42	653	0.06	0.64	0.13
log4php	92	3115	0.03	176	695	0.25	1.91	0.22
lucene	5700	23342	0.24	2574	3955	0.65	0.45	0.17
mahout	1992	8058	0.25	1013	1737	0.58	0.51	0.22
nutch	884	12251	0.07	371	1466	0.25	0.42	0.12
ode	2753	6359	0.43	2309	3465	0.67	0.84	0.54
openejb	7787	29454	0.26	7646	12217	0.63	0.98	0.41
pluto	934	11724	0.08	660	1792	0.37	0.71	0.15
solr	3595	9958	0.36	1498	2571	0.58	0.42	0.26
wicket	13680	10972	**1.25**	15837	12140	**1.30**	1.16	1.11
xerces2_j	6523	16421	0.40	3522	4540	**0.78**	0.54	0.28

Result 5: *Collaborative Groups form within ASF OSS projects organically and are unlikely to be simply an artifact of our approach. CoG members communicate and coordinate through the developer mailing lists, and their communication suggests a meaningful difference between development and coordination patterns during and outside putative CoGs. The developers surveyed describe collaboration mechanisms consistent with our findings.*

5.5.2. RQ2: Collaboration Prevalence. To contrast the time spent and code commits submitted in each project during collaboration periods as compared to periods of solitary work, we count the number

TABLE 5.7. Amount of person-days and commits in each project, separated by group vs. solitary development. This is similar to Table 5.6, only done with CoGs extracted with $\Delta t = 2$.

	Person Days (D)			Commits (C)			C/D	
	CoG	Solo	G/S	CoG	Solo	G/S	CoG	Solo
abdera	82	3394	0.02	180	1379	0.13	2.20	0.41
activemq	1178	20188	0.06	1485	5253	0.28	1.26	0.26
ant	2746	41994	0.07	4139	12384	0.33	1.51	0.29
avro	35	3335	0.01	51	978	0.05	1.46	0.29
axis2_c	1699	11602	0.15	3315	4614	**0.72**	1.95	0.40
camel	2850	18286	0.16	4043	10591	0.38	1.42	0.58
cassandra	1603	3183	0.50	2600	5035	0.52	1.62	1.58
cayenne	523	17138	0.03	986	7051	0.14	1.89	0.41
cxf	3467	30325	0.11	4940	13092	0.38	1.42	0.43
derby	2737	24812	0.11	3249	6843	0.47	1.19	0.28
hadoop_hdfs	626	9035	0.07	835	1500	0.56	1.33	0.17
harmony	200	4391	0.05	291	1154	0.25	1.46	0.26
hive	926	5975	0.15	825	1642	0.50	0.89	0.27
ivy	141	5908	0.02	248	2265	0.11	1.76	0.38
log4j	279	18770	0.01	582	3003	0.19	2.09	0.16
log4net	20	4983	0.00	29	664	0.04	1.45	0.13
log4php	20	1967	0.01	97	736	0.13	4.85	0.37
lucene	908	27707	0.03	1444	4638	0.31	1.59	0.17
mahout	399	9651	0.04	521	2041	0.26	1.31	0.21
nutch	99	13036	0.01	132	1557	0.08	1.33	0.12
ode	622	7855	0.08	1283	4162	0.31	2.06	0.53
openejb	1580	35576	0.04	4435	14409	0.31	2.81	0.41
pluto	228	12429	0.02	325	1998	0.16	1.43	0.16
solr	613	12610	0.05	828	2952	0.28	1.35	0.23
wicket	4483	20169	0.22	10669	16795	0.64	2.38	0.83
xerces2_j	1578	21366	0.07	2277	5595	0.41	1.44	0.26

of commits and days that each developer contributed/spent while inside a CoG and compare those to the commits and days that developer contributed/spent while developing individually. Then, we aggregate the results for each project. This gives us a glimpse into the amount of time spent in each project by all developers, Table 5.6, Columns 1 to 3, and the amount of commits contributed in each project, by all developers, Table 5.6, Columns 4 to 6. There we see that only in 6 out of 26 projects, on average, developers spend a comparable ($> 70\%$) or significantly higher amount of time working in groups than working alone. And in 9 out of 26 projects, on average, there are comparable ($> 70\%$) commits during group collaboration than solitary work. From the above, we can also measure the number of commits per unit time, and compare them during periods of collaboration and periods of personal work, in Table 5.6, Columns 7 and 8. Interestingly, while most projects expressed a smaller portion of commits and person days assigned to groups, the ratio

TABLE 5.8. Amount of person-days and commits in each project, separated by group vs. solitary development. This is similar to Table 5.6, only done with CoGs extracted with $\Delta t = 5$.

	Person Days (D)			Commits (C)			C/D	
	CoG	Solo	G/S	CoG	Solo	G/S	CoG	Solo
abdera	212	3992	0.05	287	1310	0.22	1.35	0.33
activemq	3219	18848	0.17	2479	4578	0.54	0.77	0.24
ant	9071	35669	0.25	6146	11042	0.56	0.68	0.31
avro	208	3759	0.06	140	975	0.14	0.67	0.26
axis2_c	3815	9486	0.40	4101	3776	**1.09**	1.07	0.40
camel	6410	14726	0.44	6271	9258	0.68	0.98	0.63
cassandra	3536	1273	**2.78**	3919	4361	**0.90**	1.11	*3.43**
cayenne	1586	16418	0.10	1922	6538	0.29	1.21	0.40
cxf	8369	25472	0.33	6985	11821	0.59	0.83	0.46
derby	11121	17823	0.62	4604	5996	**0.77**	0.41	0.34
hadoop_hdfs	2823	6838	0.41	1027	1488	0.69	0.36	0.22
harmony	405	5091	0.08	409	1046	0.39	1.01	0.21
hive	3223	3678	**0.88**	1096	1576	**0.70**	0.34	*0.43**
ivy	481	5568	0.09	459	2190	0.21	0.95	0.39
log4j	811	18242	0.04	871	2845	0.31	1.07	0.16
log4net	35	4968	0.01	35	658	0.05	1.00	0.13
log4php	92	3115	0.03	170	699	0.24	1.85	0.22
lucene	3318	25723	0.13	2271	4146	0.55	0.68	0.16
mahout	1299	8751	0.15	894	1829	0.49	0.69	0.21
nutch	502	12633	0.04	291	1501	0.19	0.58	0.12
ode	1775	6760	0.26	1963	3697	0.53	1.11	0.55
openejb	5324	31917	0.17	6795	12817	0.53	1.28	0.40
pluto	638	12020	0.05	547	1874	0.29	0.86	0.16
solr	2155	11398	0.19	1325	2674	0.50	0.61	0.23
wicket	11006	13646	**0.81**	14700	13268	1.11	1.34	0.97
xerces2_j	4327	18617	0.23	3168	4865	0.65	0.73	0.26

of commits/(person days) is greater in CoGs in all but 4 of the projects. Thus, we observe that although the amount of time and effort spent during collaboration differs from one project to another, and in most cases it is not a big portion, there seems to be an increase in productivity during these collaborations. This difference between productivity within a CoG against individual development exists and is even stronger when we lower the *temporal proximity parameter* of our CoG extraction mechanism (Tables 5.7 and 5.8).

To study whether tenure, or time spent with the project, affects developers' inclination towards co-development, we select developers with at least 2 years of commit activity (the dates of their commits span more than 2 years). We call them *dedicated* developers. We then measure the number of groups each of these developers participated in co-development during their first and last years as developers. We also

TABLE 5.9. The correlation between tenure and co-development. "Group Count" columns "First" and "Last" show the number of CoGs a dedicated developer has participated in during their first and last developer years. "F>L" shows the number of dedicated developers who have participated in more CoGs in their first year than their last."Group Days" columns are similar, but show the days a dedicated developer has been part of a CoG.

	Developers		Group Count			Group Days		
	All	Dedicated	First	Last	F>L	First	Last	F>L
abdera	7	2	18	0	2	729	0	2
activemq	25	13	300	127	7	4032	3899	6
ant	37	18	465	47	15	5636	3797	13
avro	10	1	14	9	1	365	308	1
axis2_c	20	10	450	128	8	3363	3311	4
camel	30	13	546	414	10	4857	4518	6
cassandra	13	2	83	170	0	734	734	1
cayenne	18	10	154	29	9	3381	1139	10
cxf	38	18	1160	587	13	6366	6310	13
derby	29	15	580	386	11	4599	5170	7
hadoop_hdfs	23	1	15	8	1	336	381	0
harmony	16	3	0	53	0	0	869	0
hive	18	2	91	99	1	749	731	2
ivy	6	4	33	15	2	1033	836	3
log4j	13	9	88	41	6	2830	1381	7
log4net	5	3	4	2	1	852	672	1
log4php	6	2	4	3	1	730	623	2
lucene	32	15	199	147	9	4977	4460	4
mahout	13	5	51	90	2	1558	1743	1
nutch	16	8	61	19	6	2731	1888	6
ode	15	3	134	65	3	1017	727	1
openejb	35	17	233	225	8	5495	4105	11
pluto	21	6	39	34	4	1794	1523	4
solr	18	9	188	112	7	2900	2358	5
wicket	24	12	1049	248	11	4779	4169	9
xerces2_j	26	13	379	173	10	4790	4168	10
Total	514	214	6338	3231	148	70633	59820	129

count the number of days they spent in CoGs during the first and last years. The results in Table 5.9 show that over all projects, most developers ($148/214 = 69\%$) participate in more groups in their first year. We also see that over all projects, the difference in the days participating in CoGs in the first vs. the last year ($129/214 = 60\%$) is slightly above indifference (50%). This suggests that developers are not less inclined to collaborate later on, but instead they participate in slightly fewer, longer lasting groups. One explanation for this is that, perhaps, the eagerness of youth over time gets exchanged for higher focus, or maybe loyalty, in latter years.

Result 6: *Developers spend a considerable fraction of their development time working in groups, and while their patterns of contribution and effort within CoGs vary over projects, in aggregate when in CoGs, they submit more code per unit of time. In the developers' latter time with a project they collaborate for almost the same amount of time, but participate in fewer groups.*

5.5.3. RQ3: Focus vs. Collaboration. To study the correlation between group collaboration and developer and package focus, we measure the focus of all developers and on all packages in each project.

First, we investigate the correlation between developer focus and CoG membership. Ideally, to compare more and less focused developers, we would select from the first and last quartiles in each project. However, since many of the projects have a small number of members, these quartiles would have 3 or fewer members, thereby precluding meaningful statistical analyses. Instead, we pool over all projects, *i.e.,* we take the first, Q_1, and last, Q_4, quartiles, with respect to focus, of developers from each project, and we merge the developers from all projects into two quartile lists. Thus, the first group are developers whose DAF is less than 25% of the population in their project, *i.e.,* the less focused group, and the second group are those whose DAF is greater than 75% of their projects' population, *i.e.,* the focused group.

For each group we measure the number of days the developers were part of a putative CoG. A beanplot [**172**] of the distribution of group membership days for these groups is shown in Figure 5.7. The data shows that the number of days spent in putative CoGs is negatively correlated with developer focus, *i.e.,* the more focused one is, the less likely one is to be part of a group, and vice versa. A *Mann-Whitney U test* of these two populations shows there is a statistical difference between the two (with a p-value of $< 10^{-3}$) and that we are not mislead by their visual difference.

We repeat the same process for packages, separating them and then merging them into two groups: the first group is made up of packages with MAF less than 25% of packages in their project and the second, packages with MAF greater than 75% of packages within their project. A beanplot of the distribution shows that there is also a negative effect between MAF and being in a putative CoG, *i.e.,* the higher a package's attention focus, the less likely it is being collaborated on in a group. We also confirmed this difference by running a similar Mann-Whitney U test on package focus groups (p-value $< 10^{-10}$).

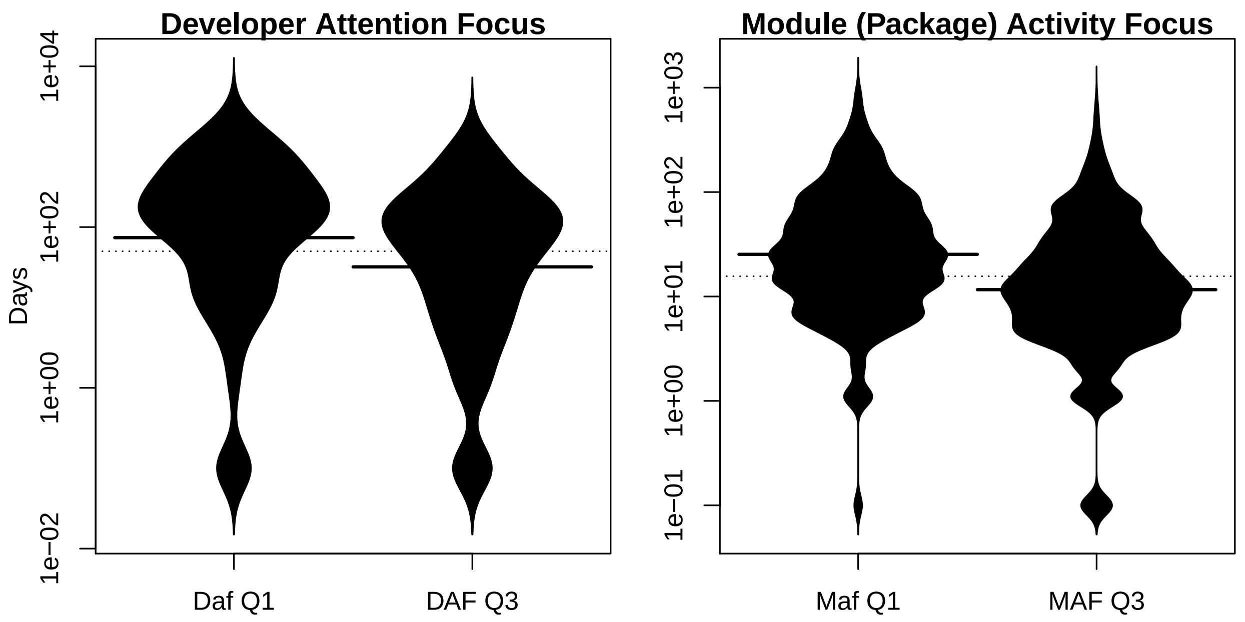


FIGURE 5.7. Distribution of collaboration time vs. focus. Left: Distribution of collaboration days for less focused vs. focused developers. Right: Distribution of days involved in collaboration for packages with high vs. low ownership.

Result 7: *Developers with more focus collaborate less often, and packages with higher ownership are also less likely to be subject to collaboration.*

5.5.4. RQ4: Productivity and Effort vs. Collaboration. We study productivity using two of the metrics previously used by Xuan and Filkov [**107**]; code effort and code growth. Code effort is defined as the number of lines added plus number of lines deleted from each package, at each commit. Code growth is number of lines added minus number of lines deleted from each package, at each commit, *i.e.,* it shows a package's size change in terms of LOC per commit.

In each commit a certain number of packages are changed; some number of lines are added and some are deleted. We group commits into those during group collaboration and those during personal development but we only count commits from developers that at some point are part of a CoG, as we wish to measure the difference in their productivity during and outside collaborations. We then measure code effort and growth on all developers in each project, for both groups of commits. If there is a meaningful statistical difference between the two populations, then developer productivity has changed during group collaboration. The results are presented in Table 5.10. The values in columns $1-4$ represent the confidence interval of the

TABLE 5.10. Correlation between co-development and developer productivity. The first 4 columns shows the difference between group and solitary development in terms of LOC per commit per file. Values greater than 3 lines have been highlighted as a significant difference. The values are rounded, so a negative 0 means the changes were negative, but close to zero. Empty cell demonstrate lack of statistical significance.

	Add	Del	Effort	Grow	Age	Msgs.	Files	Cmts.	Devs.
abdera	-0	0	-2	-2	2111	2921	3193	1492	13
activemq	-1	1	**-25**	-2	2283	20896	16788	6124	28
ant		0	**-9**	-0	4447	20433	11620	14710	44
avro	-1	0	2	-3	1074	7114	3021	984	12
axis2_c	**-6**	1	**-18**	**-5**	2995	15201	10262	6742	24
camel	**-28**	1	**-25**	**-33**	1749	27431	36965	12257	31
cassandra	**-46**	1	**-52**	**-55**	1121	4105	17125	5968	13
cayenne	-0	0	**-15**	-2	2189	6247	31489	7514	20
cxf	3	-0	1	**11**	2052	7952	37867	15322	45
derby	-0	-0	-1	0	2781	74850	6563	8301	35
hadoop_hdfs	-2	1	**-10**	-3	998	2608	1153	1529	25
harmony	**-39**	0	**-39**	**-48**	2378	31855	14898	1377	25
hive	-3	1	**-4**	**-5**	528	11373	7333	1715	18
ivy	1	0			397	907	3513	2347	9
log4j	-0	1	**-8**	-1	4114	5826	5519	3377	18
log4net					2975	1199	1060	683	7
log4php	-2	1	-3	**-11**	2263	1175	1409	803	9
lucene	-0	-0	**-28**	-1	3846	66817	6674	5343	41
mahout	-2	1	-3	**-4**	1519	18640	5123	2280	15
nutch	**-5**	2	**-4**	**-15**	2594	13041	3072	1600	16
ode	**-4**	1	**-8**	**-10**	2225	7823	11006	4901	17
openejb	1	-0	**-23**	**13**	2080	8191	43960	16620	38
pluto	-0	0	**-6**	-0	2907	4038	5971	2150	24
solr	-2	0	**-17**	**-6**	1548	21359	8534	3354	19
wicket	0		**-17**	0	1999	12479	48045	24059	24
xerces2_j	1		2	1	4373	4744	3732	7336	33

Mann-Whitney U test of the two populations. The numbers are rounded into integers and values highlighted are the ones where the test returned a p-value of < 0.05.

We observe that in terms of code effort, in 17 projects developers showed a statistically meaningful decrease in code effort while within putative CoGs. The other projects developers' change was either statistically insignificant, or small in magnitude (3 or fewer lines of code). At the same time, code growth decreases among 10 projects' developers, but also increases among 2 projects' developers. Table 5.11 again shows that these results are rather stable and do not change drastically with alteration of CoG parameters.

These results are seemingly at variance with the recent results of Xuan and Filkov [**107**], where the code growth was found to be positive during collaboration, for groups of size 2. Cross-checking with their results,

TABLE 5.11. Correlation between co-development and developer productivity. This table is similar to Table 5.10, only for $\Delta t = 2$ and $\Delta t = 5$.

	$\Delta t = 2$ Days				$\Delta t = 5$ Days			
	Add	Del	Effort	Grow	Add	Del	Effort	Grow
abdera	-1	0	-3	**-4**	-1	0	-3	-2
activemq	-0	1	**-20**	-1	-0	0	**-22**	-1
ant		0	-2	0		0	**-5**	
avro	-2	1	**-6**	**-9**		0		
axis2_c	-1	0	**-5**	-0	**-4**	1	**-11**	-2
camel	**-21**	1	**-19**	**-28**	**-26**	1	**-23**	**-32**
cassandra	**-24**	1	**-34**	**-30**	**-40**	1	**-47**	**-50**
cayenne	**-3**	-0	**-27**	**-5**		-0	**-10**	
cxf	**8**	-0	2	**22**	3	-0	2	**13**
derby	-0	-0	-1	0	-0	-0	-0	0
hadoop_hdfs	-1	0	-3	-1	-2	1	**-6**	-2
harmony	**-49**	1	**-50**	**-58**	**-34**	0	**-36**	**-42**
hive	-0	1	-1	-2	-2	1	**-3**	**-4**
ivy	2	0	1		1	1		
log4j		1	**-5**	-0	-0	1	**-8**	-1
log4net							-3	
log4php	-1	2		**-4**	-2	1	-3	**-11**
lucene	-1	-0	**-17**	-1	-0	-0	**-21**	-0
mahout	-2	1	**-3**	-3	-1	2	-2	-3
nutch	**-4**	1	**-5**	**-8**	**-5**	2	-3	**-16**
ode	**-5**	1	**-6**	**-14**	-3	1	**-7**	**-10**
openejb	1	-0	**-17**	**8**	0	-0	**-22**	**4**
pluto	1	-0	-2	1	-0	0	**-5**	-0
solr	**-8**	1	**-25**	**-14**	-1	0	**-15**	-1
wicket	0	-0	**-17**	2	0	-0	**-18**	2
xerces2_j	0	-0	4	1	1	-0	3	2

we see that our results are in fact not in disagreement. They have studied code growth in 6 ASF projects, 5 of which are present in our current study. These 5 projects are “ant”, “cxf”, “derby”, “lucene”, “openejb”. What they report in terms of code growth during pair-wise co-development is compatible with our presented findings for 2 of those 5 projects as we find that same positive growth in “cxf” and “openejb”. For the other 3 projects, growth was positive in 2 of them, and insignificant in “lucene”. Fig. 1 in their paper shows that the positive growth in these two projects is much less than the growth in “cxf” and “openejb”. We note that one of the criteria for their selection of projects to study was to have a high number of developers, which may have an effect on group collaboration patterns, and may have contributed to their results and overall conclusion. Here, our experiments were much more comprehensive and less biased, and hence, we assert, are more conclusive.

In Table 5.10 we have presented a number of additional overall characteristics for all our projects. The Spearman correlation between these characteristics and code effort shows there is a significant negative correlation between code effort and a project's total number of files (-0.57). We also observe significant correlations between code growth and the number of commits and developers in a project (0.53, 0.57). These numbers hint at having more developers and activity in projects correlates positively with higher developer productivity during collaborations, while having more packages in a project may lower code effort.

5.5.4.1. *Adjusting to Collaboration.* Our results provide compelling evidence for the hypothesis that during group collaboration developers commit more often, but in smaller amounts (see Tables 5.6 and 5.10).

A plausible explanation is that smaller contributions may decrease the chances of conflicts when more people are around. We asked developers in our survey *if and how they adjust their working style when collaborating* and the most common response was that they *commit more often and in smaller sizes*, thus in agreement with our findings. This is also consistent with the author's experience in collaborative development.

While survey participants collectively believe that collaboration *increases* productivity, they also admit that *collaboration requires extra communication and coordination* (Table 5.5). Our results, that show collaborative groups being statistically unfavorable are consistent with this, and are possibly capturing the hidden costs and overhead of socio-technical collaboration.

Result 8: *Effort spent on each package, at each commit, drops during group collaboration in most projects, and increases in only a few. Increase in code growth is correlated to project's total number of commits and developers. Committing more frequently but in smaller chunks is one of developers' methods of adjusting to a collaborative environment.*

5.6. Conclusion

In this paper, we studied how to trace group collaboration in OSS. We found that the frequency of collaboration is much less than expected levels predicted by the randomized models. This hints towards developers' selectiveness in collaborative development. Larger groups are much less prevalent than smaller ones, which is expected as the costs of coordination and communication increase with group size, along with likelihood of merge-conflicts and other such difficulties. A considerable fraction of identified groups communicate and coordinate through the developer mailing lists and the content of their communications has meaningful difference compared to their messaging patterns outside of group development. Collaboration is also not a one time thing, but people focus on fewer and longer living groups over time. Increased developer focus, as expected, correlates with reduced participation in CoGs, and respectively higher code ownership correlates with reduced co-development in groups. These results have implications for practitioners as they can lead them towards more efficient and less distracting coordination practices. They can also be beneficial in encouraging new developers into collaborating with senior developers and facilitating the process.

Finally, we found that effort expended is almost universally lowered while developer groups collaborate. While the same agreement does not exist for file code growth (although more or less prevalent), developers in more populated projects expressed increased code growth during group collaboration. These results offer evidence towards collaboration being recognized as coming at a cost, but is generally beneficial towards lowering developer effort. In other words, working in groups has an increased cost, but pays dividends. Thus, the cost of scaling collaborations should be an important consideration when a developer joins new projects or teams, as it may ultimately be counterproductive for larger groups. This of course is a generic issue, very well known in management science for centrally managed teams, where dealing with team scalability and corrections vis-a-vis project management is perhaps more direct [**159**].

There are several directions for followup research to this work, including studying the trade-off between the benefits of collaboration and its costs and identifying ranges of group sizes lying in the sweet spot. In-depth interviews with developers regarding specific details of collaboration and its dynamics can be used to increase the resolution of our results.

5.7. Threats to validity

There are a number of threats to our approaches and our conclusions. First and foremost is the threat to *construct validity.* our method for recognizing groups is clearly only a second hand approximation. We argue, that if groups in fact exist, then our method must capture them, so long as they produce code at the same time. Clearly, a group of people can do work together which does not involve all of them coding at the same time. But we do not presume to be able to identify such groups, only their subsets involved in coding. The non-coding members of the group are not directly relevant to our study, so our groups may be only subsets of actual groups. Clearly, intent cannot be established from trace data, so any results must necessarily be possibly circumstantial. But by studying group prevalence over a large set of OSS projects, we hope that there will be sufficient evidence to make our results convincing.

We also recognize several other threats to the *internal validity* of our work. We used a survey as a more direct approach of establishing the existence of groups, but passing of time limited our ability to explicitly ask specifics of the team members, forcing us to settle for less precise questions and fewer answers than we wanted. The small number of developers participating in our survey also limited our ability to infer from the responses. We note that those responses still greatly agreed on most issues.

Simplifying productivity down to two measures of code growth and effort is sure to miss some dimensions of productivity, *e.g.,* code churn. On the other hand, they do capture more information than simple growth, and are a natural first order approximation measure of energy expended.

We used source code files in each project to identify groups. There are more file types in each project, such as xml files, some of which we may actually be subject to group collaboration, but we had to exclude them to remove bias introduced by files that are not handled by developers, such as “pom.xml”. The number of these files is, however, limited and statistically much smaller than the tens of thousands of source code files in the projects.

We acknowledge the threat to *external validity* of this research, as our findings are based solely on ASF projects and may be affected by cultural and environmental parameters that may limit the applicability of our findings to other OSS ecosystems such as GitHub.

CHAPTER 6

From Here, There, and Everywhere: Cross-Project Cloning in GitHub

6.1. Introduction

Programmers opportunistically reuse code, and do so frequently [**173**], as good code is valuable and so is their time. Code that works well is hard to come by; Knuth's *The Art of Computer Programming* has numerous short, elegant, jewel-like bits of code that would be hard for mere mortals to produce in a reasonable time interval. As the adage goes, *"Good programmers write good code; great programmers steal"*. In fact, often, programming of well-defined problems amount to a look-up [**174**], first in one's own, and then in others' code repositories, followed by judicious copying and pasting. The bigger the corpus of code available, the more likely it is that one will find what one is looking for.

The advent of Software forges like GitHub, and Google Code, and Q&A sites like Stack Overflow has significantly enhanced this kind of opportunistic code reuse [**175–178**]; one can readily forage online for code and ideas, and often reuse existing code. In fact, developers not only look for the opportunity to reuse code but also advertise their own high-quality code for others to reuse by paste-bin like web applications such as GitHub Gist [**179**], Pastebin [**180**], and Codeshare [**181**].

Social coding ecosystems like GitHub present a fundamental shift in reuse opportunities. With millions of available OSS projects hosted in GitHub, among which thousands of developers freely migrate, GitHub provides an excellent opportunity for collaboration and code reuse [**119**, **152**]; in fact, GitHub actively facilitates the process by providing features like advanced code search and GitHub gist. The reused code varies in size, ranging from few lines of code to methods, and even larger—one can copy and reuse an entire project or a set of files and make smaller modifications to cater local needs [**182**] [1]. Thus, ecosystem level code sharing and reuse may be different from within project code reuse—one would expect limited methods and file copies in a within project setting, resulting in a new kind of software development practices. In this paper, we focus on studying such ecosystem level code sharing and reuse across multiple GitHub projects.

[1]See StackExchange question: http://programmers.stackexchange.com/questions/-193415/best-practices-for-sharing-tiny-snippets-of-code-across-projects

An established way to study code reuse is by analyzing code clones [**183**, **184**]. "Code Clones" are repeating snippets of code that are surprisingly similar and arise from copy-pasting practice [**185**], automatic program generators [**186–188**], *etc.* Clones can even emerge independently due to the predictable syntax of underlying programming languages and API usage [**189**]. Clones were studied extensively in the literature although primarily limited to clones within project boundaries [**190**, **191**]. This is expected especially before the current era of project ecosystems, since project repositories were separated not only physically, but also logistically by naming conventions and work practices. The path of least resistance might have been to copy code *within* the same project; searching and reusing code from other projects might have been quite a challenge. In contrast, GitHub, by deliberate design, is a universal platform for maintaining repositories, with minimal boundaries between projects. Thus, in this paper, we focus on studying cross-project cloning across GitHub projects to understand the nature of code sharing and reusing in an ecosystem. In particular, we study how much of coding in a project comes from other projects via code clones, as compared to from within the project. Next, we characterize the cross-project clones *w.r.t.* their prevalence, types, and sources. To our knowledge, we are the first to study cross-project cloning in an ecosystem setting.

We selected $5,500$ Java projects from GitHub and using Deckard [**192**], an established clone detection tool, we detected both within and cross-project clones. We then only chose the clone pairs that appear across different projects and further studied them using statistical and network science approaches, and with multiple case studies. In summary, we found the following:

- Cross-project clones comprise between 10% and 30% of all code clones within projects, and up to 5% of the code base.
- Cross-project clones exist due to different reasons ranging from implementing similar functionalities to structural similarity and being auto-generated.
- A considerable proportion of all cross-project clones are *utility clones*, where entire files, directories or even projects are copied with a minimum amount of change. They serve different purposes such as exposing APIs and utility code reuse.
- Many projects are disproportionately sources for clones, and many others are sinks, *i.e.,* contain surprising number of clones from different projects. In the mean time, a majority of projects tend to take more than they give.
- Code-cloning seems to follow an onion model: most clones come from the same project, then from projects in the same application domain, and finally from projects in different domains.

- There is no supporting evidence of cloning being more prominent among projects with shared developers or among more experienced and active/diverse developers.

These results show that cross-project cloning contributes to a significant proportion to Open Source software development. Cross project clones are in general restricted to projects in a similar domain and follow certain fixed patterns. Moreover, some projects share more code than others. These findings call for new tool support to facilitate code discovery—one should prioritize the search within the same domain and to the projects that often serve as super-sources. Further, our study indicates there are certain code patterns that are commonly used across many projects and can be suitable candidates for code sharing—this observation calls for a recommending tool for paste-bin applications.

The rest of the paper is organized as follows. In Section 6.2 we motivate our research questions and follow it with a review of the existing works relevant to our study in Section 6.3. The Methodology of our work is described in Section 6.4 followed by the presentation of our results in Section 6.5. Finally, in Section 6.6 we discuss potential threats that can affect our findings and Section 6.7 we conclude.

6.2. Research Questions

As a first step towards understanding cross-project cloning, we seek to identify the rate of code reuse within GitHub ecosystem. In particular, we want to know how many clones are there, what is the rate of within vs. cross-project clones, how many lines are cloned on average, and if there is a correlation between project metrics and clone density. Once we know the extent of cross-project clones, we are curious how they look like, *i.e.,* whether certain kind of code is cloned more across the project boundaries. We ask all these in a juggernaut of a question.

Research Question 9: How prevalent are clones across GitHub ecosystem? How does that compare to within project cloning? What are the common types of code that are cloned across projects?

Once we know the extent and nature of cross-project clones, a natural question might be where are these clones originating from. It is conceivable that some projects, whether by design or not, serve more as libraries or frameworks, and thus can be good sources of cloned code, *i.e.,* their code has been reused in many other projects. Likewise, some projects like device drivers can borrow a lot of code from similar drivers and thus may be more dependent on reusing code than expected. Knowing which project has borrowed code from which other projects, we build a project clone network to help us answer the following question:

Research Question 10: Are there projects that serve as clone sources more than their shares? Are some projects reusing other project's code at a greater rate?

We further investigate the ecosystem nature of GitHub to understand where clones come from, where they end up, and if there is a social mechanism related to them. Being a platform with uniform rules, GitHub enables easy searches beyond one's own project's repository. In fact, GitHub via its gamified, uniform social coding environment, encourages people to code in multiple projects, not just over time, but also at the same time. It is not uncommon to find people who code on multiple projects in a day [**193**]. Code cloning certainly helps with speeding up coding, and perhaps makes this possible.

So, it comes naturally then to wonder if the existence of people shared between projects correlates with sharing clones among the projects, or in other words is there a congruence between the network of shared clones and the network of shared developers?

The other topic of interest is to see if clones are confined within certain functional domains. In other words, do most cross-project clones come from projects within the same domain? Or do they come from another specific or any domain? It is reasonable to think that code might be more similar between two projects from the same domain, say two projects about Web development than between projects from disparate domains, say Web and Compiler. All these lead us to the following research question:

Research Question 11: Can we find support for existing mechanisms through which cloning happens, such as being promoted by multi-project developers or being confined within application domains?

6.3. Related works

In general, code clones are considered to be similar code fragments [**191**], although the notion of *similarity* widely varies and mostly depend on underlying code clone detectors. For example, widely popular CCfinder [**191**] detects clone based on lexical and syntactic analysis, while Deckard [**192**] relies on Abstract Syntax Trees (AST) matching technique. Rattan *et al.* have studied and summarized the majority of clone detection techniques in their review [**194**].

By applying different clone detection techniques, previous studies reported as much as 10% to 30% of the code in many large scale projects is identified as clones (*e.g., gcc*-8.7%, *JDK*-29% [**191**], and *Linux*-22.7%). Gabel and Su studied source code uniqueness across a large code corpus of Source-Forge Projects [**195**] and found a lack of uniqueness in code at a granularity of 1 to 7 lines of source code. Software is not only repetitive, but also changed similarly, as found by multiple studies based on different Microsoft projects, Linux, and multiple open source Java projects [**183**, **184**, **196**]. However, none of these studies focused on cross-project code cloning in a large scale ecosystem.

The most closely related work that studied cross-project cloning is by Ossher *et al.* [**197**]; they studied similar files across 13,000 java projects from multiple ecosystems. However, they were limited to measuring only similar files as opposed to clone code fragments and did not differentiate between within and cross-project cloning. In this work we focus on both file and code fragment cloning. We further analyzed clones in multiple temporal, spatial, and developer dimensions to understand which projects in an ecosystem contributed most clones, or whether certain developers promote more clones than others.

Among the other cross-project cloning studies, Ray and Kim found evidence of significant similar changes among the BSD product family [**182**]. Al-Ekram *et al.* studied cloning in similar software and concluded that many times clones develop accidentally due to several reasons such as following protocols [**189**]. We also found evidences of accidental cloning in cross-project settings.

Nguyen *et al.* [**184**] reported up to 70-100% of repetitive source code changes across 2,841 open source Java projects, and the repetitiveness is higher and more stable in across projects vs. within projects. However, code changes are not similar to static code and thus, study of repetitive changes are different than code clone study. For example, a study of repetitive changes may ignore utility files as they are seldom changed in software evolution.

6.4. Methodology

Before diving into methodology, let us clarify the definition of clones that we employ in this paper.

Clones are pieces of code of at least N tokens that are exactly similar to each other, except for token names.

TABLE 6.1. Basic statistics of our study's dataset.

#Projects	#Developers	#Files	Total LOC	Project Age
5753	23K	1.04M	105M	1 to 6 Years

TABLE 6.2. Parsed clone information based on Deckard output.

CloneID	CloneSubID	ProjectName	FileName	From	Len.
01	01	Distrotech_cyrus-sasl	java/CyrusSasl/SaslOutputStream.java	48	32
01	02	mb-linux-msli	uClinux-dist/lib/libcyrussasl/java/CyrusSasl/SaslOutputStream.java	48	32
02	01	encog-java-examples	src/main/java/org/encog/examples/neural/gui/ocr/Entry.java	200	32
02	02	encog-java-workbench	src/main/java/org/encog/workbench/tabs/query/ocr/DrawingEntry.java	194	32
03	01	agit	agit-test-utils/src/main/java/com/madgag/agit/matchers/GitTestHelper.java	46	3
03	02	nexus	gateway/src/main/java/org/isolution/nexus/xml/soap/SOAPMessageUtil.java	50	3
03	03	nexus	gateway/src/main/java/org/isolution/nexus/xml/soap/SOAPMessageUtil.java	89	3

The reason that we define codes as exactly similar with no room for variation is that we want to study intentional clones rather than accidental ones [**189**] as much as possible, so we choose this definition to eliminate many such occurrences. We are aware that this is a rather strict definition, specially with larger choices of N it excludes many potential clones, but instead it increases our confidence that the clones reported are more likely to be intentional rather than a result of accidents and simply following protocol.

Our methodology involved several steps including project selection, repository cloning, clone mining, domain analysis and finally, statistical analysis of the results.

6.4.1. Project Selection. We used the January 4th 2014 dump of GHTorrent database [**198**] to mine the list of users, commit history and other meta-data about projects. Using this information we selected projects in Java that had at least 2 developers and 1 year old and contained more than 10 commits. These criteria remove much smaller and younger projects with single developers that usually do not contribute to such ecosystem study but skew the results [**199**].

We also eliminated projects that are forked using GitHub interfaces, as it would highly skew our findings while implicating a meaningless duplicate code. After these filtering, we were able to identify $8,599$ projects that matched our criteria and had not been deleted by the time of our data gathering (Sept. 2015). After running the initial clone detection algorithm as described later in the paper, we are left with $5,753$ projects where non-trivial clones are found. An overview of the size and scope of our dataset is shown in Table 6.1.

6.4.2. Identifying Project Domains. To identify the domain that each project belongs to, we employed a tool developed by Ray *et al.* [**200**]. This tool iteratively uses Latent Dirichlet Allocation (LDA), a popular topic analysis algorithm on projects' readme text and description to identify the topic of each project. For each topic, we then manually inspect the most frequent keywords representing it and find an appropriate domain name to describe the topic.

6.4.3. Identifying Clones. We used Git to gather the repositories for all the mentioned projects, and rolled back each repository to the January 4th 2014. We then used Deckard [**192**] to identify clones present in this collection. We chose Deckard because of its performance and also having access to its authors that provided us the chance to fully utilize the tool's potential and get more accurate results.

In Deckard, you need to set 3 Parameters to specify clones and their accuracy, *"Stride"*, *"Similarity"*, and *"Token Size"*. We set Stride to *infinite* and Similarity to 1 to find the most similar and non-spurious clones. For the choice of *Token Size* which specifies the minimum size of a piece of code (in tokens) to be accepted as clones, we chose three values: 20, 30 and 50. With these choices, we are now also able to study the effect of this parameter on our findings and observe how various clone sizes differ in occurrence. Although Deckard is very efficient, it has a minor shortcoming that it only works on a single project. To bypass this issue and identify cross-project clones, we had to place all the mentioned projects' repositories in one *umbrella* directory, with one folder per project and then treat this as one great project. Once the cloning results were gathered, we used the directory information of each file in the results to indicate whether files containing clones are in the same project or not and label them as cross-project clones.

After parsing Deckard's output, the results are stored in a table similar to Table 6.2. We call a clone a *cross-project clone*, if for the corresponding cloneID, at least two of its instances come from two different projects. In Table 6.2 all three clones are considered as cross-project clones. We call a clone a *Within-project clone* if least two of its instances are from the same project. A clone can be both cross-project and within-project; we call them as *Hybrid clones* (*e.g.,* cloneID 3 in Table 6.2).

6.4.4. Extracting Utility Clones. During our case-studies of cross-project clones, we noticed multiple clone instances where entire files or even directories or in extreme cases an entire project was copied from one place to another, with or without minor changes. Such cloned artifacts often serve as utility or library code and we call them as *utility clones*. We extracted such utility clones based on their file names and directory structures—they often share similar nomenclature as reported by Ossher *et al.* [**197**]. For example, we identified files `engine/src/main/java/org/json/JSONArray.java` and `src/org/json/JSONArray.java` of projects HomeSnap and ModDamage as utility clones. While these cases

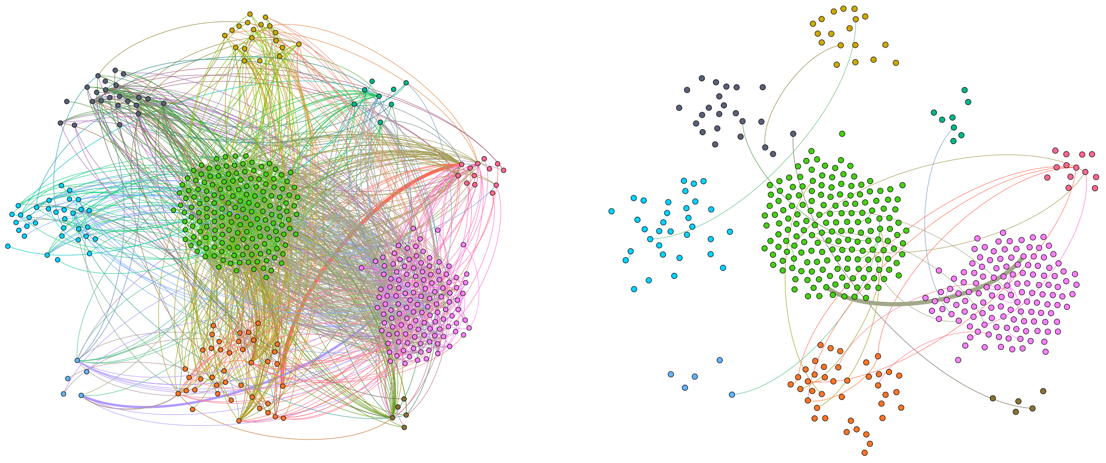


FIGURE 6.1. Left: The co-clone graph for 50 Token clones. The coloring corresponds to project domains. Right: The co-developer graph for $\theta = 5\%$ and induced by the co-clone graph.

are indeed examples of cross-project code reuse, their borrowing dynamics arguably differs from that of regular clones. We have extracted them from the rest of clones to studied them separately.

6.4.5. Constructing the Ecosystem Graphs. We construct two graphs from the dataset we have gathered. The first is a *co-clone graph* which shows which projects share code clones. The second, *co-developer graph* highlights developers active between multiple projects.

To construct the co-clone graphs, we first create an empty graph with each project as a node. We then add an edge between two projects if they contain instances of the same clone, and if such an edge already exists, we increment its weight by 1. When constructing the co-clone graph a disambiguation problem arises. Ideally, we want an edge between each pair of projects to imply one borrowed code from another, and in the case a cloneID has only 2 instances, it is simple. If there are more than 2 instances of a clone, how should we infer the edges? Do we add edges between all the project instances? We chose to identify the oldest clone instance using `git blame` as the *source* and add an edge all other projects to the project with oldest instance. This way, all projects point to the "original" source of information, like a star topology. While the validity of our approach can be debated, there is no exact way of knowing what the "real" source of a piece of code is. Thus, any other approach can be criticized the same way. Figure 6.1 shows the co-clone graph for clones with token size 50.

TABLE 6.3. The resulting size of co-developer graphs with varying θ. Values greater than 5% filter out too many edges.

	θ	20	30	50
No. of Edges	0	91703	79384	552
	5	8304	7623	60
	10	4562	4170	35
	20	2074	1885	12
	30	984	910	4
No. of Nodes		5643	5282	435

We construct the co-developer graphs in a slightly different manner. Again, we have a graph with projects as nodes, but here, for each developer that is "active" across K projects, we create a *K-clique* among those projects. A major threat to this approach is that often, developers are not active in all the projects they participate in to the same extent; they may be heavily invested in some, and occasionally or even rarely contribute to the rest. We address this issue by defining a contribution threshold θ. We consider developer d as a contributor to project p if the ratio of his/her contribution (in terms of number of commits) to project p to his/her overall contribution to all the projects P is at least θ. In other words:

$$\frac{NCom_{d,p}}{\sum_{p_i \in P} NCom_{d,p_i}} \geq \theta$$

We have constructed the co-developer graphs for the following values of θ: $0\%, 5\%, 10\%, 20\%, 30\%$. To compare co-clone graphs with co-developer graphs, we *induce* the co-developer graph by the co-clone graph. Table 6.3 shows the resulting graph sizes for our choices of θ. Figure 6.1 shows the induced co-developer graphs for θ=5%.

6.5. Results and Discussion

Prior to analyzing characteristics of code reuse across GitHub, we begin with a straightforward question that investigates the extent cross-project clones in the ecosystem, namely:

RQ1: Cloning Prevalence in GitHub. Table 6.4 presents a summary of the extent of clone code in selected GitHub projects. In total, we find evidences of 354268, 243150, and 13908 unique clones for token sizes 20, 30, and 50 respectively. Code clones make up between 5% to 10% of the entire code base of these projects and their average size ranges from 5 to 15 LOC. Code is cloned in three ways within and across GitHub projects:

TABLE 6.4. **Extent of Cloning across GitHub**

	Token Size		
	20	30	50
Projects with Clones	5753	5533	628
#Unique clones	354,268	243,150	13,908
#Unique within-project clones	246,390	170,729	12,664
	(69%)	(70%)	(91%)
#Unique cross-project clones	121,565	59,589	1,037
	(34%)	(25%)	(7%)
#Unique hybrid clones	53,512	21,276	389
#Unique utility clones	39,825	34,108	596
	(11%)	(14%)	(4%)
Size of projects with clone *	104.85	103.39	13.27
Size of clones	9.86	8.31	0.67
Size of within-project clones	7.64	6.17	0.59
Size of cross-Project clones	4.85	2.93	0.1
Size of utility clones	0.93	1.09	0.04
Mean within-project clone size †	5.63	9.51	15.6
Mean cross-project clone size	5.21	9.63	20.8
Mean utility clone size	9.91	13.59	23.89
Median within-project clone size	4	7	11
Median cross-project clone size	4	6	13
Median utility clone size	6	8	12

* Total size of projects and clones are in Million LOCs (MLOC).
† Mean and median size of clones are in LOC.

(i) Within-project Clones. These are most common source of code cloning (69% to 91%) and widely studied in the past [**183**, **191**, **195**]. In this paper, we will not discuss such clones in details as our focus is mostly on cross-project cloning.

(ii) Cross-project Clones. We find a non-trivial amount of cloned code across multiple projects. At a smaller token size of 20, nearly (34%) of all identified clones come from other projects. As the token size increases, the extent of cross-project clones drops to nearly 7% at token size 50. This suggests that larger chunks of code are less likely to be copied from other projects. The median size of within-project and cross-project clones are statistically identical for 20 and 30 token clones but they start to diverge in 50 token clones, with cross-project clones being considerably larger than their counterparts.

We further check how cross-project clone prevalence depends on project size, *i.e.,* whether larger projects have more cross-project clones. For each project, we measure 2 metrics as a proxy for clone prevalence: total number of cloned lines and total number of files having cloned code. Figure 6.2 shows the distribution of cloned lines *w.r.t.* project size measured in LOC and top subfigure of Figure 6.3 plots cloned files vs. total files. While the first plot shows cloned lines slowly increases as the project size grows with a sublinear trend, the latter plot shows number cloned files increases linearly as the number of files

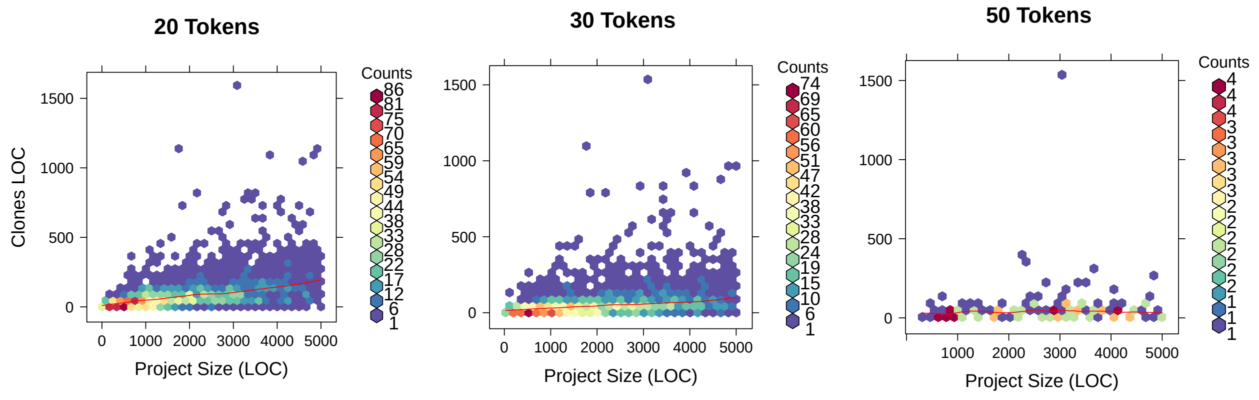


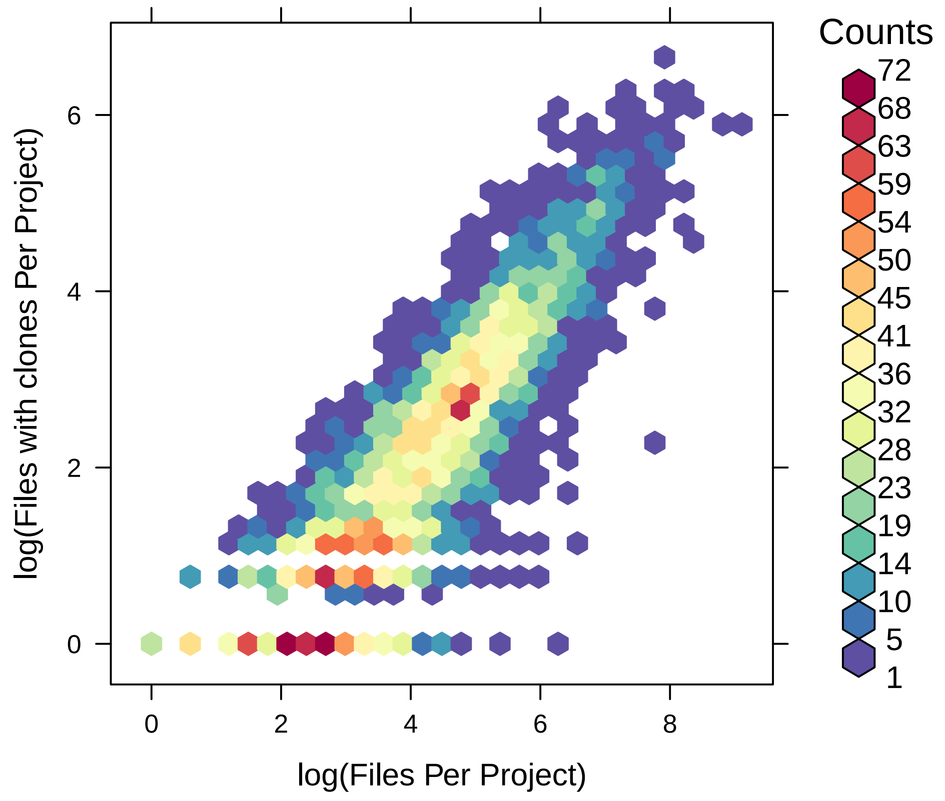
FIGURE 6.2. Size of cloned lines in a project vs. projects' size (LOC); we observe a sub-linear trend.

increases in a project. Since project size is highly correlated with total number of files, such a trend can only be explained if clones are non-uniformly distributed across files. To confirm that bottom subfigure of Figure 6.3 presents a histogram of clone density (= number of cloned lines *w.r.t.* total lines) per file across all the studied projects. Here we ignore the files that do not have a single cloned lines as they might have skewed the distribution. The histogram is heavily skewed with a peak at 2.5% clone density confirming that cross-project clone density varies significantly across files. This observation is true for clones across all the token sizes.

TABLE 6.5. Cross-Project Cloning: a case study of 90 uniquely cloned instances

Type of cross-project clones	**count**	**patterns** (individual count)
(a) Code fragments implementing similar functionalities	34 (38%)	add new objects to object queue (2), wrapper methods to call set of other methods (2), wrapper methods to null-check on method arguments or create the argument object or initialize the arguments (7), iterate over an array to access a particular element (2), Object serialization (3), Putting elements in HashMap (2), Read IO (2), Recursively calling similar function (1), Release resources/-clean up routine (2), Initializing memory elements (2), Similar API call (4), Testing object initialization (1), Similar exception handling routine (4)
(b) Methods implementing similar functionalities	25 (28%)	similar function in related files (18), similar functions in different files (7)
(c) Clones due to structural similarity	29 (32%)	iterating in loop structure (6), array initialization (3), if-else structure (8), calling methods with similar signature (6), similar class definition (3), long return statement (3)
(d) Autogenrated files	2 (2%)	WSDL files (1), ANTLR Translator Generator (1)

To understand the nature of cross-project clones, we manually studied 90 unique clone instances, chosen randomly, that are detected at token size 30 (see Table 6.5). 66% of these clones appear as developers often reuse code to implement similar functionalities, either my reusing smaller code fragments (38%) social-cloning/or cloning an entire method body (28%) (rows a and b in Table 6.5). For example, we find 7 unique



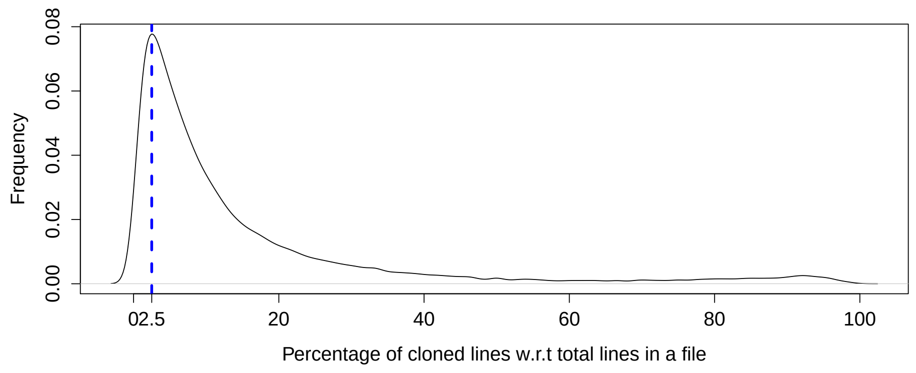


FIGURE 6.3. Top: Files having cross-project clones vs. total files per project showing a linear trend. Bottom: Percentage of cloned lines per file showing a heavily skewed trend with a peak frequency at clone density 2.5%. Both the figures consider clones at toke size 30.

clone instances where developers perform operations like null checks and object initializations on a method's arguments before calling the method. There are other cases where a similar code fragment is reused for adding new objects to an object queue, putting elements in a hash map, performing object serializations (*e.g.,* `toString` method), calling APIs in similar ways, *etc.* Such clones are closely tied with data structure/API usage. Developers also clone entire methods to implement higher level functionalities (row b). Such cloned methods can appear either in related (file/class names or directory structures resemble each other) but non-identical files or in apparently unrelated files. For instance, a method `connectSocket` that implements socket connection routine in Android applications for a given IP address and port, is implemented identically in three files AgeClient.java, FakeSocketFactory.java, EasySSLSocketFactory.java of projects App-Growth-Engine-Android-SDK, ReGalAndroid, and cmsandroid respectively. Such semantic clones can be good candidates for pastebin like GitHub Gist [**179**] where people frequently share their commonly used code snippets for future use. In fact, the above example of `connectSocket` method is shared by developer darrikmazey in his GitHub gist `https://gist.github.com/darrikmazey/9521134`. Further, (c) 32% clones across different projects show structural similarities without much of a deeper semantic resemblance. Iterating over while/for loops, similar if-else structures, array initializations are few common examples in this category. Finally, we find (d) two instances of clones where the whole files are auto generated.

Note that, some of the similar code patterns may exist both within and across multiple projects. We call such clones as "hybrid" clones. In our study, we find 53k, 21k, and 389 unique hybrid clones for different token sizes.

(iii) Utility Clones. We also notice a substantial amount of clones where the whole files or modules are reused with little or no modifications. As shown in Table 6.4, such clones contribute to 11%, 14% and 4% of all the unique cloned instances at token sizes 20, 30, and 50 respectively. We find that such files/modules are usually part of utility or library code and developers often reuse them across multiple projects. We collectively call such clones as utility clones. Note that the utility clones are also instances of cross-project clones where instead of copying code fragments developers reuse entire files or modules across projects. To investigate in details, we manually studied 264 of such files that are cloned up to 9 projects and find five types of utility clones (see Table 6.6): (a) cloning library source code to expose APIs: Developers often copy the entire source code of popular libraries to use their APIs. For example, we find several copies of ActionBarSherlock, an Android library for using action bar design patterns. (b) Cloning Files/Modules to reuse utility code: Developers often clone single utility files or modules to reuse some common functionalities such as HTML or JSON file parser. (c) We find several instances of Java standard

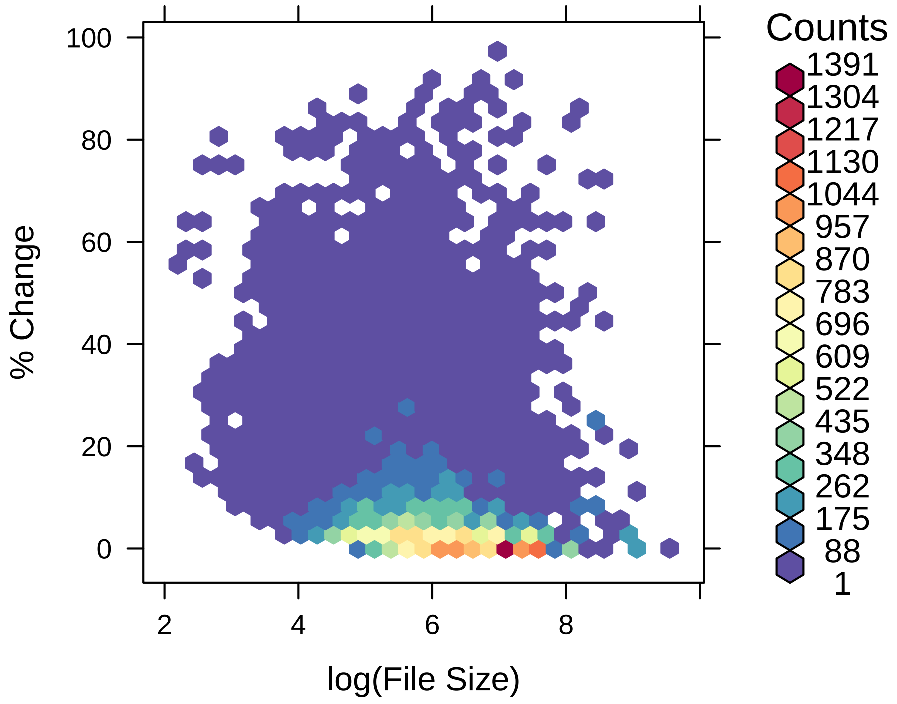


FIGURE 6.4. The distribution of change percentage vs file size in utility clones at toke size 30.

library (*e.g.,* java.io, java.lang, and java.util code) code reused multiple times across different projects. (d) Another common source of utility clones is extensions of Java by third party vendors like Test Driven Development (TDD) in Action, *etc.* (e) There are some other sources of utility clones come from related projects, example or demo code, *etc.* Ossher et al. [**197**] also reported similar types of file-level cloning activities in SourceForge, Apache, Java.net and Google Code projects; thus, confirming our findings.

TABLE 6.6. Utility Cloning: a case study of 264 cloned files

Type of Utility Clones	Examples
(a) Cloning library source code to expose APIs	Mobile development APIs like ActionbasSharlock, Game development APIs like Forestry, Security protocol related APIs like SSL
(b) Cloning Files/Modules to reuse utility code	Binary file processing utility, Android ListView, Date and Time Processing utility, Encoder/Decoder, IO, HTML/JSON parser, utility for network protocols like DNS, HTTP, etc.
(c) Java Standard Library	java.io, java.lang, java.util code, etc.
(d) Java Extensions	Java extensions by some third party vendor like Test Driven Development (TDD) in Action , etc.
(e) Other	Related/Duplicate projects, Example Code/Demo etc.

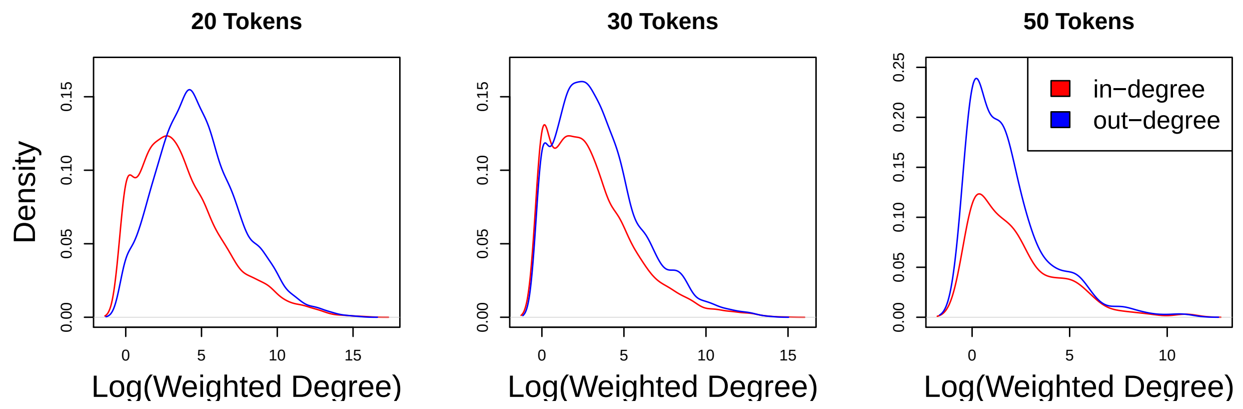


FIGURE 6.5. The weighted in-degree (indicating clone provider) and weighted out-degree (indicating clone forage) distribution of co-clone graphs with different token sizes. Each node represents a project, and edge exists if two projects share a clone. The weight of the edge represents number of unique clones between two project nodes. We observe an overall rightwards shift of the peak in the out-degree (*i.e.,* forage) indicating most projects forage clones than provide them.

Once cloned in different projects, the utility files are seldom changed. In fact, among all the cloned files at token size 30, 56.12% files were never modified, and 75% files are changed only up to 2.94% *w.r.t.* respective file sizes. For the utility files that differ across multiple instances by at least one line, Figure 6.4 plots the percentage of changed lines *w.r.t.* total file size. The brighter region at the bottom in the hexbinplot indicates that even such files are limited to a small amount of changes, with the median of 3.95%. These results show that utility clones are mostly static across different projects.

Result 9: *A significant amount of code is reused across GitHub projects, and the corresponding clones follow some fixed patterns.*

In the rest of this work, we only present the results of cross-project clones. We also repeated the experiments including utility clones. Overall results were substantially similar.

Now that we have seen cross-project cloning is quite prevalent across GitHub projects, we wonder from where are these clones coming from. This leads us to the following Research Question:

RQ2: Sources of Clones. Before checking whether there are some common sources of cross-project clones, we investigate whether cloning is a uniform and undirected process *i.e.,* do projects *provide* as much clones as they *forage*. Figure 6.5 shows the distribution of weighted in-degrees (indicating provide) and out-degrees (indicating forage) of projects in co-clone graphs, as described in Section 6.4.5. We observe an overall rightwards shift of the peak in the out-degree. We know that for directed graphs, sum of in-degrees and out-degrees are equal:

$$\sum_{v \in V} deg^{+}(v) = \sum_{v \in V} deg^{-}(v)$$

Thus, a peak in the out-degree (forage) implies a heavier tail in in-degree (provide) distribution. This indicates, in general, most projects forage code more than provide it. The heavier tail in in-degree distribution also implies that there are few projects that act as a *super-source* of clones with much higher in-degree than the top out-degrees.

TABLE 6.7. Projects with highest weighted in-degree and out-degree in co-clone graphs. Project names are trimmed to a maximum of 16 characters to fit on page.

	20		30		50	
Rank	IN (Provide)	OUT (Forage)	IN (Provide)	OUT (Forage)	IN (Provide)	OUT (Forage)
1	acceleo	SIREn	acceleo	android_platform	nuxeo-features	GoodData-CL
2	com.idega.block.	EclipsePlugin	android-sdk	wl-rwiki	slps	Henshin-Editor
3	xDoc	RobotML-SDK	rwiki	ecl	OpenFaces	cropinformatics-
4	mvdetsen	jnr-x86asm	is.idega.idegawe	EclipsePlugin	plexus-utils	b3log-latke
5	com.idega.core	OpenFaces	jdnssec-dnsjava	gatein-shindig	ActionBarSherloc	plexus-container
6	jedit-ruby-plugi	ecl	shindig	mahout	amplafi-json	andlytics
7	VUE	android_platform	mahout-commits	log4jna	MSMB	coverity-plugin
8	rwiki	wl-rwiki	log4j	Henshin-Editor	jbidwatcher	spring-ide
9	android-sdk	osate2-ocarina	android_packages	osate2-ocarina	wl-calendar	WISE-Portal
10	luaj	Henshin-Editor	com.idega.block.	jgit	jbosstools-base	HomeSnap

To investigate this further, Table 6.7 shows the 10 highest ranking projects in terms of weighted in-degree (provide) and out-degree (forage). As expected, we observe that the top 10 lists in all three cases completely differ, *i.e.,* major clone providers are different from the greatest foragers. For example, at token size 30, projects like `acceleo`, `android-sdk`, and `rwiki` are top provider of clones while projects `android_platform`, `wl-rwiki`, and `ecl` are top foragers.

We conclude that cloning is not an uniform process and most projects forage more clones than provide them; thus the projects that serve as clone super-source are important for maintaining the GitHub ecosystem.

Result 10: *Cross-system cloning is a directed and non-uniform phenomenon and most projects "forage" clones than "provide" them. There are also projects that serve as hubs or "super-sources" of clones to other projects.*

RQ3: Mechanisms of Cloning. Next, we explore the existence of several potential mechanisms of cloning within GitHub, namely cloning within the boundaries of domains, cloning among neighboring projects, and finally, cloning by experienced and active members.

Cloning within Domain Boundaries. We selected cross-project clones and checked whether the projects which share clones are within the same domain or not. The results are presented in Table 6.8 which shows that *cross-domain* clones outnumber *within-domain* clones by almost 2 to 1.

TABLE 6.8. The statistics of cross-domain and within-domain clones based on the co-clone graphs

	20	30	50
All Edges	808623	244948	4011
Within-Domain Edges	281700	96466	1400
Cross-Domain Edges	526923	148482	2611
Within/Cross Ratio	0.53	0.65	0.54

We further investigate this issue and quantify the frequency of clones across different domains. For each domain we normalize the number of clones it shares with other domains (including itself) by the size of both domains *i.e.,* the number of projects in that domain:

$$Clones(D_i, D_j) = \frac{\sum_{P_k \in D_i, P_l \in D_j} Clones(P_k, P_l)}{|D_i| \times |D_j|}$$

Here $Clones(P_k, P_l)$ denotes the number of clones between projects P_k and P_l, and $|D_i|$ denotes the number of projects in domain D_i[2]. Such normalization is important because this will remove the bias of larger domains having a larger code base, and hence large amount of clones (see Figure 6.2). Finally, for each domain, we measure the percentage of clones that come from within and other domains. We present the results in Figure 6.6.

Despite the initial appearance of Table 6.8, we see that once we control for domain size, there is this clear pattern that shows a greater concentration of clones within domains rather than across them. Of course there are a few exceptions in the heat maps, especially at token size 50, but the overall trend is clear. We conclude that cross-project clones are more likely to happen within the boundaries of domains than across.

Cloning Among Neighboring Projects. A possible mechanism through which code is cloned is within the set of a project's neighbors. In this context two projects are considered neighbors if there is at least one

[2]This number is among the set of the projects that had any clones at all. So the total sum of all domain sizes adds up to the first row numbers of Table 6.4.

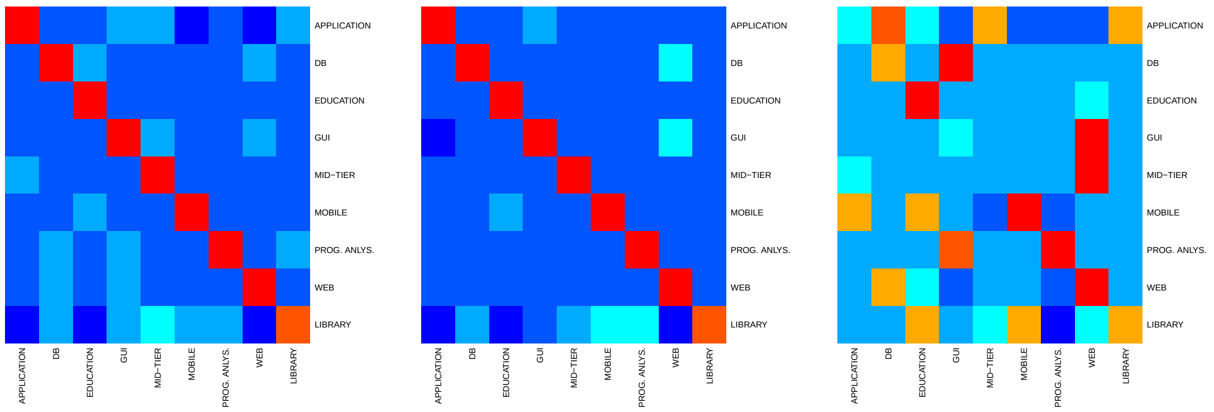


FIGURE 6.6. The heat map of Cross-domain clone frequency for token size 20, 30 and 50 from left hand side to right hand side respectively. Each entry shows the percentage of cross-domain clones that appear between the corresponding domains normalized by the size of both domains. Hotter colors (red, orange) indicate a higher value and colder colors such as deep blue indicate lower values.

developer that is actively participating in both projects. So in essence we want to see if projects tend to clone across the edges in the co-developer graph or not.

To study the overlap of the co-clone and co-developer networks we utilize the *congruence* metric employed by Xuan *et al.* [**108**]. This measure simply represents the sum of weighted edges that are common between the two networks divided by the sum of all the weighted edges in one of them. Thus, it is an *asymmetric* measure:

$$Congruence(A, B) = \frac{\sum_{E_i \in A \cap B} Weight(E_i)}{\sum_{E_i \in A} Weight(E_i)}$$

We observed a little to no congruence *w.r.t.* co-clone graphs ($Cong < 0.07$), but we did observe some congruence between *w.r.t.* co-developer graphs ranging between 0.16 and 0.25 depending on token size and other parameters such as contribution threshold θ (see Section 6.4.5). We compared this against a baseline distribution obtained through evaluating the congruence between randomized co-clone graphs and co-developer graphs and found that our results are not significantly different from the baseline distribution. Thus, we conclude that most likely cross-project cloning does not depend on project neighborhoods.

Cloning by Senior and Expert Developers. We investigated the correlation between development team size and clone density which is depicted in Figure 6.7. A weak sublinear trend exists here, so the clone

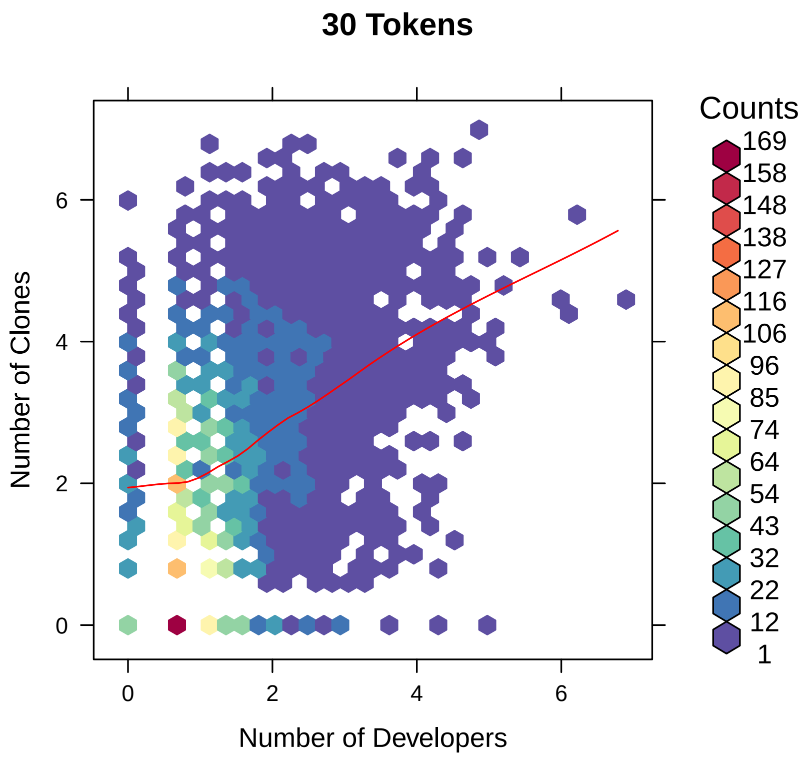


FIGURE 6.7. Cross-project Clone density vs number of developers in each project for 30 token clones. Both axes are logged. We observe a rather superlinear trend of clones increasing with number of developers.

density slowly increases as the number of developers in a project increase. But are all developers equally participating in cloning, or are some more active than others, and can we find out who?

To study whether certain attributes of developers would be indicative of their rate of code cloning we gathered the following measures for each developer who was the author of at least one clone: (i) *Number of Clones.* The total number of distinct clones that was authored by the developer, (ii) *Clone Size.* The total size (in LOC) of all clone instances written by her, (iii)*Number of Projects.* The number of projects the developer has participated in, (iv) *Age.* The first date she has contributed to any of the mentioned projects, (v) *Number of commits.* The number of times she has committed to the mentioned projects.

First we measure the spearman correlation between these features across our dataset. As we see in Table 6.9, even with low p-values, the correlations are small or average at best. Still we attempted to go further and predict the number of clones using other parameters. We used linear regression with the following formula:

$$\#Clones \sim \#Projects + Age + \#Commits$$

While the p-value for coefficients were again very low ($p\text{-}value < 0.001$), the *coefficient of determination i.e.,* adjusted R-squared was 0.07 which hints that our predictors are not good at describing the outcome. We tweaked the formula through several ways, such as replacing any of the variables with the *log* of its value, and replacing the output variable with size of clones, but none of these improved the coefficient of determination dramatically, with the highest outcome being below 0.09. We conclude that developers clone at different rates, there is no immediate relationship between seniority, activity, or expertise, and the frequency of cloning, at least not a linear or log-linear relationship, with only these parameters. There may be other parameters that when combined would provide further insight into this matter.

TABLE 6.9. Correlation between developer features. Number & size of clones are based on 20 token clones. The results were almost identical for larger token sizes. for all the values here, we have $p\text{-}value < 0.001$.

	Clone Size	#Projects	#Commits	Age
#Clones	0.94	0.18	0.41	0.20
Clone Size		0.17	0.39	0.20
#Projects			0.51	0.33
#Commits				0.33

.

Result 11: *Cross-project cloning is more prolific within a domain boundary. Authorship of these clones does not play a critical factor in such cloning.*

6.6. Threats to Validity

We recognize several threats to the validity of our work:

(i) Threats to Construct Validity. Our strict definition of clones may have resulted in lower reported rates of cloning. We deliberately made this decision so as to exclude *accidental clones* as much as possible. The identification of utility clones was solely based on nomenclature, and may have missed a number of such clones with different file names. However, our results point towards entire files being copied and such renaming being rare. Domain assignments may not be perfect, especially since some projects could conceptually belong to several domains.

During the construction of co-clone graphs we assumed all copies are “foraged” from the first/oldest source, and that may not be the case due to several reasons such as discovery limitations, personal preference, and even an existing source outside of the scope of our dataset. While this is a valid threat, to our knowledge there is no information available that would help in identifying the “real” source, and the only solution would be a direct query from the developers which is practically impossible.

(ii) Threats to Internal Validity. In our search for evidence of cloning mechanisms, we were unable to find any support for certain mechanisms, but this may be due to unknown factors at play that we were unable to capture in our dataset. Another issue is that the classification of clones in the case study is prone to personal interpretation. To address this issue we used the feedback of an external developer to validate our categorization.

(iii) Threats to External Validity. Our study discusses cross-project cloning within one ecosystem (GitHub) and several parameters such as platform culture and facilities may make our findings unique to this platform and less applicable to others. At the same time, we only studied Java projects, and while Java is second-most popular language in GitHub (after JavaScript), different programming languages, specially non-object-oriented ones such as Perl or Lisp may exhibit different patterns of cloning. Being aware of this issue, We have limited the scope of our study to Java projects. At the same time, we intend to extend this study with inclusion of several other languages such as Python and C.

6.7. Conclusion

In this paper we studied the patterns of cloning code in the GitHub ecosystem. We discover that cross-projects clones exist and are rather prevalent in GitHub. We also see that there are projects that act as sources or sinks and that most projects forage more clones than they provide. Finally we investigated for evidence in support of cloning mechanisms. We found support for cloning withing domains that supports the “onion model” hypothesis. We also did not find evidence of developer activity and experience affecting their cloning tendencies, but as the saying goes “absence of evidence is not evidence of absence”.

Not only our findings shed light into this phenomenon and provide a better understanding of cloning, but we argue that our results are actionable in that it can help towards a “discovery tool” that would facilitate “foraging” code for resuse. For example, our results hints towards prioritizing within-domain projects and super-sources in the search for the desired snippets.

APPENDIX A

Developer Questionnaire

The questionnaire is sent to each individual through email. Each email starts with a proper introduction of the authors, and our research. afterwards, they are asked to complete the form and submit it to us.

ASF Collaborative Development Questionnaire

* Required

How would you describe your involvement in this project?

e.g., project founder, core developer, ...

How frequently did/do you work on this mentioned project? *

○ Daily

○ Once per 2-3 days

○ Once per week

○ Less than once per week

What are some typical tasks you carried out in this project? Please give a few examples.

e.g., fixing bugs, implementing a new feature, ...

How do you choose which tasks to work on? Do you choose your own tasks? How do you prioritize which tasks to work on first?

How long did tasks you worked on typically take, from start to finish? *

If you were part of a bigger task, please answer with the overall task in mind

○ 1-2 days
○ 3-5 days
○ A week
○ 2 weeks
○ Other:

When does work by others influence you / your work directly? *

When it is in the same files you are touching at the time; the same packages; the whole project or something else

○ The file(s) I am working on
○ The package(s) I am working on
○ The whole project
○ Other:

Which of your tasks do you consider to be more collaborative than the others?

e.g., bug fixes, adding new features, ...

How many people do collaborative tasks typically involve?

○ 2
○ 3
○ 4
○ 5
○ 6

○ more

How do you coordinate your work with collaborators on the same task? What communication channels do you use?

Do you discuss with them prior to task assignment, during task work, or after task completion?

How do you adjust your working style when collaborating as opposed to during solitary work, if at all?

e.g., by committing less frequently, or by pushing smaller commits more frequently, ...

When is it beneficial and when is it detrimental to collaborate with others on the same task?

Please tell us how much you agree or disagree with the following sentences *

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Working on the project was a collaborative effort	○	○	○	○	○
You actively attempted to “team up” with others to complete tasks	○	○	○	○	○
Collaboration increases productivity	○	○	○	○	○
Collaboration increases merge conflicts and introduces some difficulties	○	○	○	○	○
Collaboration requires extra coordination and communication	○	○	○	○	○

Bibliography

[1] The open source definition. http://opensource.org/docs/osd. (2012)

[2] "Usage share of operating systems," https://en.wikipedia.org/wiki/Usage_share_of_operating_systems#Market_share_by_category, Accessed: 2016-03-20.

[3] "Masquerade Attack," https://www.techopedia.com/definition/4020/masquerade-attack, Accessed: 2016-03-20.

[4] M. Schonlau, W. DuMouchel, W.-H. Ju, A. F. Karr, M. Theus, and Y. Vardi, "Computer intrusion: Detecting masquerades," Statistical science, 2001, pp. 58–74.

[5] H.-S. Kim and S.-D. Cha, "Empirical evaluation of svm-based masquerade detection using UNIX commands," Computers & Security, vol. 24, no. 2, 2005, pp. 160 – 168.

[6] R. Maxion and T. Townsend, "Masquerade detection using truncated command lines," in Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on, 2002, pp. 219–228.

[7] B. Szymanski and Y. Zhang, "Recursive data mining for masquerade detection and author identification," in Information Assurance Workshop, 2004. Proceedings from the Fifth Annual IEEE SMC, June 2004, pp. 424–431.

[8] J. Seo and S. Cha, "Masquerade detection based on svm and sequence-based user commands profile," in Proceedings of the 2Nd ACM Symposium on Information, Computer and Communications Security, ser. ASIACCS '07, 2007, pp. 398–400.

[9] M. Latendresse, "Masquerade detection via customized grammars," in Detection of Intrusions and Malware, and Vulnerability Assessment, ser. Lecture Notes in Computer Science, 2005, vol. 3548, pp. 141–159.

[10] A. Mahajan, "Masquerade detection based on unix commands," master thesis, San Jose State University, 2012.

[11] S. Greenberg, "Using unix: Collected traces of 168 users," Advanced Technologies, The Alberta Research Council, 1988, pp. 1–13.

[12] S. Fitchett and A. Cockburn, "Accessrank: Predicting what users will do next," in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ser. CHI '12. New York, NY, USA: ACM, 2012, pp. 2239–2242. [Online]. Available: http://doi.acm.org/10.1145/2207676.2208380

[13] R. Chinchani, A. Muthukrishnan, M. Chandrasekaran, and S. Upadhyaya, "Racoon: rapidly generating user command data for anomaly detection from customizable template," in Computer Security Applications Conference, 2004. 20th Annual, Dec 2004, pp. 189–202.

[14] D. M. German, "The GNOME project: a case study of open source, global software development," Software Process: Improvement and Practice, vol. 8, no. 4, 2003, pp. 201–215.

[15] K. Crowston, K. Wei, J. Howison, and A. Wiggins, "Free/libre open-source software development: What we know and what we do not know," ACM Computing Surveys (CSUR), vol. 44, no. 2, 2012, p. 7.

[16] A. Mockus, R. T. Fielding, and J. D. Herbsleb, “Two case studies of open source software development: Apache and Mozilla,” ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 11, no. 3, 2002, pp. 309–346.

[17] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye, “Evolution patterns of open-source software systems and communities,” in IWPSE. ACM, 2002, pp. 76–85.

[18] Y. Ye and K. Kishida, “Toward an understanding of the motivation of open source software developers,” in ICSE. IEEE, 2003, pp. 419–429.

[19] G. Hertel, S. Niedner, and S. Herrmann, “Motivation of software developers in Open Source projects: an internet-based survey of contributors to the Linux kernel,” Research Policy, vol. 32, no. 7, 2003, pp. 1159–1177.

[20] G. Robles and J. M. Gonzalez-Barahona, “Contributor turnover in libre software projects,” in Open Source Systems. Springer, 2006, pp. 273–286.

[21] N. Ducheneaut, “Socialization in an open source software community: A socio-technical analysis,” CSCW, vol. 14, no. 4, 2005, pp. 323–368.

[22] B. S. Butler, “Membership size, communication activity, and sustainability: A resource-based model of online social structures,” Information systems research, vol. 12, no. 4, 2001, pp. 346–362.

[23] B. Kogut and A. Metiu, “Open-source software development and distributed innovation,” Oxford Review of Economic Policy, vol. 17, no. 2, 2001, pp. 248–264.

[24] J. Roberts, I. Hann, and S. Slaughter, “Understanding the motivations, participation, and performance of open source software developers: A longitudinal study of the Apache projects,” Management science, vol. 52, no. 7, 2006, pp. 984–999.

[25] R. Fielding, “Shared leadership in the Apache project,” Communications of the ACM, vol. 42, no. 4, 1999, pp. 42–43.

[26] V. Sinha, S. Mani, and S. Sinha, “Entering the circle of trust: developer initiation as committers in open-source projects,” in MSR. ACM, 2011, pp. 133–142.

[27] C. Jensen and W. Scacchi, “Role migration and advancement processes in OSSD projects: A comparative case study,” in ICSE. IEEE, 2007, pp. 364–374.

[28] C. Bird, A. Gourley, P. Devanbu, A. Swaminathan, and G. Hsu, “Open borders? pImmigration in open source projects,” in MSR. IEEE, 2007, pp. 6–6.

[29] M. Cataldo, J. D. Herbsleb, and K. M. Carley, “Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity,” in ESEM. ACM, 2008, pp. 2–11.

[30] N. Bettenburg and A. E. Hassan, “Studying the impact of social structures on software quality,” in ICPC. IEEE, 2010, pp. 124–133.

[31] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy, “Does distributed development affect software quality?an empirical case study of Windows Vista,” Communications of the ACM, vol. 52, no. 8, 2009, pp. 85–93.

[32] Y. Long and K. Siau, “Social network structures in open source software development teams,” Journal of Database Management (JDM), vol. 18, no. 2, 2007, pp. 25–40.

[33] K. Crowston and J. Howison, “The social structure of free and open source software development,” First Monday, vol. 10, no. 2, 2005.

[34] C. De Souza, J. Froehlich, and P. Dourish, “Seeking the source: software source code as a social and technical artifact,” in SIGGROUP. ACM, 2005, pp. 197–206.

[35] E. Raymond, “The cathedral and the bazaar,” Knowledge, Technology & Policy, vol. 12, no. 3, 1999, pp. 23–49.

[36] K. Stewart and S. Gosain, “An exploratory study of ideology and trust in open source development groups,” in ICIS. ACM, 2001, pp. 1–6.

[37] M. Newman, S. Forrest, and J. Balthrop, “Email networks and the spread of computer viruses,” Physical Review E, vol. 66, no. 3, 2002, pp. 035 101(R):1–4.

[38] C. Fershtman and N. Gandal, “Direct and indirect knowledge spillovers: the “social network” of open-source projects,” The RAND Journal of Economics, vol. 42, no. 1, 2011, pp. 70–91.

[39] G. Krogh and E. Hippel, “The promise of research on open source software,” Management Science, vol. 52, no. 7, 2006, pp. 975–983.

[40] W. Scacchi, “Free/Open source software development: Recent research results and methods,” Advances in Computers, vol. 69, 2007, pp. 243–295.

[41] 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. ACM, 2006, pp. 137–143.

[42] F. Rahman, D. Posnett, and P. Devanbu, “Recalling the imprecision of cross-project defect prediction,” in Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. ACM, 2012, p. 61.

[43] F. Rahman, D. Posnett, I. Herraiz, and P. Devanbu, “Sample size vs. bias in defect prediction,” in Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM, 2013, pp. 147–157.

[44] M. Jureczko and L. Madeyski, “Towards identifying software project clusters with regard to defect prediction,” in Proceedings of the 6th International Conference on Predictive Models in Software Engineering. ACM, 2010, p. 9.

[45] M. Jureczko and D. Spinellis, “Using object-oriented design metrics to predict software defects,” Models and Methods of System Dependability. Oficyna Wydawnicza Politechniki Wrocławskiej, 2010, pp. 69–81.

[46] D. Posnett, V. Filkov, and P. Devanbu, “Ecological inference in empirical software engineering,” in Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society, 2011, pp. 362–371.

[47] G. Von Krogh, S. Spaeth, and K. Lakhani, “Community, joining, and specialization in open source software innovation: a case study,” Research Policy, vol. 32, no. 7, 2003, pp. 1217–1241.

[48] I. Herraiz, G. Robles, J. Amor, T. Romera, and J. González Barahona, “The processes of joining in global distributed software projects,” in International Workshop on Global Software Development for the Practitioner. ACM, 2006, pp. 27–33.

[49] B. Shibuya and T. Tamai, “Understanding the process of participating in open source communities,” in International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development. IEEE, 2009, pp. 1–6.

[50] I. Qureshi and Y. Fang, “Socialization in open source software projects: A growth mixture modeling approach,” Organizational Research Methods, vol. 14, no. 1, 2011, pp. 208–238.

[51] M. Zhou and A. Mockus, “What make long term contributors: Willingness and opportunity in OSS community,” in ICSE. IEEE, 2012, pp. 518–528.

[52] R. A. Depue and P. F. Collins, “Neurobiology of the structure of personality: Dopamine, facilitation of incentive motivation, and extraversion,” Behavioral and Brain Sciences, vol. 22, no. 03, 1999, pp. 491–517.

[53] W. Schultz, “Behavioral theories and the neurophysiology of reward,” Annu. Rev. Psychol., vol. 57, 2006, pp. 87–115.

[54] R. E. Lucas, E. Diener, A. Grob, E. M. Suh, and L. Shao, “Cross-cultural evidence for the fundamental features of extraversion.” Journal of personality and social psychology, vol. 79, no. 3, 2000, p. 452.

[55] M. C. Ashton, K. Lee, and S. V. Paunonen, “What is the central feature of extraversion? social attention versus reward sensitivity.” Journal of personality and social psychology, vol. 83, no. 1, 2002, p. 245.

[56] S. Deterding, M. Sicart, L. Nacke, K. O’Hara, and D. Dixon, “Gamification. using game-design elements in non-gaming contexts,” in CHI. ACM, 2011, pp. 2425–2428.

[57] R. Cheng and J. Vassileva, “Design and evaluation of an adaptive incentive mechanism for sustained educational online communities,” User Modeling and User-Adapted Interaction, vol. 16, no. 3-4, 2006, pp. 321–348.

[58] R. Farzan, J. M. DiMicco, D. R. Millen, C. Dugan, W. Geyer, and E. A. Brownholtz, “Results from deploying a participation incentive mechanism within the enterprise,” in CHI. ACM, 2008, pp. 563–572.

[59] A. Anderson, D. Huttenlocher, J. Kleinberg, and J. Leskovec, “Steering user behavior with badges,” in WWW. ACM, 2013, pp. 95–106.

[60] S. Grant and B. Betts, “Encouraging user behaviour with achievements: an empirical study,” in MSR. IEEE, 2013, pp. 65–68.

[61] A. Begel and B. Simon, “Novice software developers, all over again,” in Proceedings of the Fourth international Workshop on Computing Education Research. ACM, 2008, pp. 3–14.

[62] G. Dai and K. P. De Meuse, “A review of onboarding literature,” Lominger Limited, Inc., a subsidiary of Korn/Ferry International, 2007.

[63] T. N. Bauer and B. Erdogan, “Organizational socialization: The effective onboarding of new employees.” 2011.

[64] T. N. Bauer, T. Bodner, B. Erdogan, D. M. Truxillo, and J. S. Tucker, “Newcomer adjustment during organizational socialization: a meta-analytic review of antecedents, outcomes, and methods.” Journal of applied psychology, vol. 92, no. 3, 2007, p. 707.

[65] B. Vasilescu, A. Serebrenik, M. Goeminne, and T. Mens, “On the variation and specialisation of workload—a case study of the GNOME ecosystem community,” Empirical Software Engineering, 2013, pp. 1–54.

[66] E. Kouters, B. Vasilescu, A. Serebrenik, and M. G. J. van den Brand, “Who’s who in GNOME: Using LSA to merge software repository identities,” in ICSM. IEEE, 2012, pp. 592–595.

[67] M. Goeminne and T. Mens, “A comparison of identity merge algorithms for software repositories,” Science of Computer Programming, vol. 78, no. 8, 2013, pp. 971–986.

[68] A. Guzzi, A. Bacchelli, M. Lanza, M. Pinzger, and A. v. Deursen, “Communication in open source software development mailing lists,” in MSR. IEEE, 2013, pp. 277–286.

[69] N. Bettenburg, E. Shihab, and A. E. Hassan, "An empirical study on the risks of using off-the-shelf techniques for processing mailing list data," in ICSM. IEEE, 2009, pp. 539–542.

[70] B. Vasilescu, A. Serebrenik, P. T. Devanbu, and V. Filkov, "How social Q&A sites are changing knowledge sharing in open source software communities," in CSCW. ACM, 2014, pp. 342–354.

[71] J. Cohen, Applied multiple regression/correlation analysis for the behavioral sciences. Lawrence Erlbaum, 2003.

[72] Q. Vuong, "Likelihood ratio tests for model selection and non-nested hypotheses," Econometrica: Journal of the Econometric Society, 1989, pp. 307–333.

[73] H. B. Mann, "Nonparametric tests against trend," Econometrica: Journal of the Econometric Society, 1945, pp. 245–259.

[74] W. S. Cleveland, "Robust locally weighted regression and smoothing scatterplots," Journal of the American statistical association, vol. 74, no. 368, 1979, pp. 829–836.

[75] D. Spencer, Card sorting: Designing usable categories. Rosenfeld Media, 2009.

[76] J. Scott, Social network analysis. Sage, 2012.

[77] Q. Xuan, F. Du, and T.-J. Wu, "Empirical analysis of internet telephone network: From user id to phone," Chaos: An Interdisciplinary Journal of Nonlinear Science, vol. 19, no. 2, 2009, p. 023101.

[78] L. d. F. Costa, F. A. Rodrigues, G. Travieso, and P. R. Villas Boas, "Characterization of complex networks: A survey of measurements," Advances in Physics, vol. 56, no. 1, 2007, pp. 167–242.

[79] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D.-U. Hwang, "Complex networks: Structure and dynamics," Physics reports, vol. 424, no. 4, 2006, pp. 175–308.

[80] R. Albert, H. Jeong, and A.-L. Barabási, "Error and attack tolerance of complex networks," nature, vol. 406, no. 6794, 2000, pp. 378–382.

[81] P. Holme, B. J. Kim, C. N. Yoon, and S. K. Han, "Attack vulnerability of complex networks," Physical Review E, vol. 65, no. 5, 2002, p. 056109.

[82] R. E. Mirollo and S. H. Strogatz, "Synchronization of pulse-coupled biological oscillators," SIAM Journal on Applied Mathematics, vol. 50, no. 6, 1990, pp. 1645–1662.

[83] A. Arenas, A. Díaz-Guilera, J. Kurths, Y. Moreno, and C. Zhou, "Synchronization in complex networks," Physics Reports, vol. 469, no. 3, 2008, pp. 93–153.

[84] W. Yu, G. Chen, and J. Lü, "On pinning synchronization of complex dynamical networks," Automatica, vol. 45, no. 2, 2009, pp. 429–435.

[85] C. P. Ayala, D. S. Cruzes, O. Hauge, and R. Conradi, "Five facts on the adoption of open source software," Software, IEEE, vol. 28, no. 2, 2011, pp. 95–99.

[86] S. K. Sowe, I. Stamelos, and L. Angelis, "Understanding knowledge sharing activities in free/open source software projects: An empirical study," Journal of Systems and Software, vol. 81, no. 3, 2008, pp. 431–446.

[87] R. Sen, S. S. Singh, and S. Borle, "Open source software success: Measures and analysis," Decision Support Systems, vol. 52, no. 2, 2012, pp. 364–372.

[88] D. S. Pattison, C. A. Bird, and P. T. Devanbu, "Talk and work: a preliminary report," in Proceedings of the 2008 international working conference on Mining software repositories. ACM, 2008, pp. 113–116.

[89] C. Y. Baldwin and K. B. Clark, Design rules: The power of modularity. MIT press, 2000, vol. 1.

[90] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," Software Engineering, IEEE Transactions on, vol. 31, no. 10, 2005, pp. 897–910.

[91] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now?: an empirical study of bug characteristics in modern open source software," in Proceedings of the 1st workshop on Architectural and system support for improving software dependability. ACM, 2006, pp. 25–33.

[92]

[93] F. A. Haight and F. A. Haight, "Handbook of the poisson distribution," 1967.

[94] A.-L. Barabasi, "The origin of bursts and heavy tails in human dynamics," Nature, vol. 435, no. 7039, 2005, pp. 207–211.

[95] R. D. Malmgren, D. B. Stouffer, A. E. Motter, and L. A. Amaral, "A poissonian explanation for heavy tails in e-mail communication," Proceedings of the National Academy of Sciences, vol. 105, no. 47, 2008, pp. 18 153–18 158.

[96] Y. Wu, C. Zhou, J. Xiao, J. Kurths, and H. J. Schellnhuber, "Evidence for a bimodal distribution in human communication," Proceedings of the national academy of sciences, vol. 107, no. 44, 2010, pp. 18 803–18 808.

[97] P. R. Cohen and H. J. Levesque, Teamwork. SRI International Menlo Park, 1991.

[98] E. O. Wilson, "What is sociobiology?" Society, vol. 15, no. 6, 1978, pp. 10–14.

[99] E. E. Salas and S. M. Fiore, Team cognition: Understanding the factors that drive process and performance. American Psychological Association, 2004.

[100] M. Di Penta, M. Harman, G. Antoniol, and F. Qureshi, "The effect of communication overhead on software maintenance project staffing: a search-based approach," in Software Maintenance, 2007. ICSM 2007. IEEE International Conference on. IEEE, 2007, pp. 315–324.

[101] J. Child, "Organizational structure, environment and performance: the role of strategic choice," Sociology, vol. 6, no. 1, 1972, pp. 1–22.

[102] N. Nohria and R. Eccles, Networks and organizations: structure, form, and action. Harvard Business School Press, 1994.

[103] J. R. Katzenbach, The wisdom of teams: Creating the high-performance organization. Harvard Business Press, 1993.

[104] L. A. Dugatkin, Cooperation among animals. Oxford Series in Ecology and Evolution, 1997.

[105] K. Crowston, Q. Li, K. Wei, U. Y. Eseryel, and J. Howison, "Self-organization of teams for free/libre open source software development," Information and software technology, vol. 49, no. 6, 2007, pp. 564–575.

[106] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu, "Latent social structure in open source projects," in Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering. ACM, 2008, pp. 24–35.

[107] Q. Xuan and V. Filkov, "Building it together: Synchronous development in OSS," in Proceedings of the 34th International Conference on Software Engineering. ACM, 2014.

[108] Q. Xuan, A. Okano, P. Devanbu, and V. Filkov, "Focus-shifting patterns of oss developers and their congruence with call graphs," in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2014, pp. 401–412.

[109] M. Gharehyazie, D. Posnett, B. Vasilescu, and V. Filkov, "Developer initiation and social interactions in oss: A case study of the apache software foundation," Empirical Software Engineering, 2014, pp. 1–36.

[110] Q. Xuan, H. Fang, C. Fu, and V. Filkov, "Temporal motifs reveal collaboration patterns in online task-oriented networks," Physical Review E, vol. 91, no. 5, 2015, p. 052813.

[111] Q. Xuan, M. Gharehyazie, P. T. Devanbu, and V. Filkov, "Measuring the effect of social communications on individual working rhythms: A case study of open source software," in Social Informatics (SocialInformatics), 2012 International Conference on. IEEE, 2012, pp. 78–85.

[112] Q. Xuan, P. T. Devanbu, and V. Filkov, "Converging work-talk patterns in online task-oriented communities," arXiv preprint arXiv:1404.5708, 2014.

[113] Q. Xuan and V. Filkov, "Synchrony in social groups and its benefits," in Handbook of Human Computation. Springer, 2013, pp. 791–802.

[114] F. Rahman and P. Devanbu, "Ownership, experience and defects: a fine-grained study of authorship," in Proceedings of the 33rd International Conference on Software Engineering. ACM, 2011, pp. 491–500.

[115] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code!: examining the effects of ownership on software quality," in Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. ACM, 2011, pp. 4–14.

[116] J. Whitehead, I. Mistrík, J. Grundy, and A. van der Hoek, "Collaborative software engineering: concepts and techniques," in Collaborative Software Engineering. Springer, 2010, pp. 1–30.

[117] I. Mistrík, J. Grundy, A. Van der Hoek, and J. Whitehead, "Collaborative software engineering: challenges and prospects," in Collaborative Software Engineering. Springer, 2010, pp. 389–403.

[118] A. Avritzer and D. J. Paulish, "A comparison of commonly used processes for multi-site software development," in Collaborative Software Engineering. Springer, 2010, pp. 285–302.

[119] W. Scacchi, "Collaboration practices and affordances in free/open source software development," in Collaborative software engineering. Springer, 2010, pp. 307–327.

[120] Y. Lin, "Hybrid innovation: The dynamics of collaboration between the floss community and corporations," Knowledge, Technology & Policy, vol. 18, no. 4, 2006, pp. 86–100.

[121] F. Lanubile, C. Ebert, R. Prikladnicki, and A. Vizcaíno, "Collaboration tools for global software engineering," IEEE software, no. 2, 2010, pp. 52–55.

[122] J. D. Herbsleb and D. Moitra, "Global software development," Software, IEEE, vol. 18, no. 2, 2001, pp. 16–20.

[123] H. Holmstrom, E. Ó. Conchúir, P. J. Ågerfalk, and B. Fitzgerald, "Global software development challenges: A case study on temporal, geographical and socio-cultural distance," in Global Software Engineering, 2006. ICGSE'06. International Conference on. IEEE, 2006, pp. 3–11.

[124] A. Sarma, J. Herbsleb, and A. Van Der Hoek, “Challenges in measuring, understanding, and achieving social-technical congruence,” in Proceedings of Socio-Technical Congruence Workshop, In Conjuction With the International Conference on Software Engineering, 2008.

[125] M. Grechanik, J. A. Jones, A. Orso, and A. van der Hoek, “Bridging gaps between developers and testers in globally-distributed software development,” in Proceedings of the FSE/SDP workshop on Future of software engineering research. ACM, 2010, pp. 149–154.

[126] J. D. Herbsleb, A. Mockus, T. A. Finholt, and R. E. Grinter, “An empirical study of global software development: distance and speed,” in Proceedings of the 23rd international conference on software engineering. IEEE Computer Society, 2001, pp. 81–90.

[127] B. Al-Ani and H. K. Edwards, “A comparative empirical study of communication in distributed and collocated development teams,” in Global Software Engineering, 2008. ICGSE 2008. IEEE International Conference on. IEEE, 2008, pp. 35–44.

[128] S. Jalali and C. Wohlin, “Global software engineering and agile practices: a systematic review,” Journal of software: Evolution and Process, vol. 24, no. 6, 2012, pp. 643–659.

[129] E. Carmel, Global software teams: collaborating across borders and time zones. Prentice Hall PTR, 1999.

[130] E. Carmel and R. Agarwal, “Tactical approaches for alleviating distance in global software development,” Software, IEEE, vol. 18, no. 2, 2001, pp. 22–29.

[131] C. Ebert and P. De Neve, “Surviving global software development,” Software, IEEE, vol. 18, no. 2, 2001, pp. 62–69.

[132] I. Omoronyia, J. Ferguson, M. Roper, and M. Wood, “A review of awareness in distributed collaborative software engineering,” Software: Practice and Experience, vol. 40, no. 12, 2010, pp. 1107–1133.

[133] J. D. Herbsleb, “Global software engineering: The future of socio-technical coordination,” in 2007 Future of Software Engineering. IEEE Computer Society, 2007, pp. 188–198.

[134] Y. Takhteyev and A. Hilts, “Investigating the geography of open source software through github,” 2010.

[135] T. Nguyen, T. Wolf, and D. Damian, “Global software development and delay: Does distance still matter?” in Global Software Engineering, 2008. ICGSE 2008. IEEE International Conference on. IEEE, 2008, pp. 45–54.

[136] K. Nakakoji, Y. Ye, and Y. Yamamoto, “Supporting expertise communication in developer-centered collaborative software development environments,” in Collaborative Software Engineering. Springer, 2010, pp. 219–236.

[137] D. Redmiles, A. Van Der Hoek, B. Al-Ani, T. Hildenbrand, S. Quirk, A. Sarma, R. Filho, C. de Souza, and E. Trainer, “Continuous coordination-a new paradigm to support globally distributed software development projects,” Wirtschafts Informatik, vol. 49, no. 1, 2007, p. 28.

[138] A. Sarma, B. Al-Ani, E. Trainer, R. S. Silva Filho, I. A. da Silva, D. Redmiles, and A. van der Hoek, “Continuous coordination tools and their evaluation,” in Collaborative Software Engineering. Springer, 2010, pp. 153–178.

[139] D. Damian and D. Moitra, “Guest editors’ introduction: Global software development: How far have we come?” Software, IEEE, vol. 23, no. 5, 2006, pp. 17–19.

[140] D. Šmite, C. Wohlin, T. Gorschek, and R. Feldt, “Empirical evidence in global software engineering: a systematic review,” Empirical software engineering, vol. 15, no. 1, 2010, pp. 91–118.

[141] D. Šmite, C. Wohlin, R. Feldt, and T. Gorschek, “Reporting empirical research in global software engineering: A classification scheme,” in Global Software Engineering, 2008. ICGSE 2008. IEEE International Conference on. IEEE, 2008, pp. 173–181.

[142] M. Pinzger and H. C. Gall, “Dynamic analysis of communication and collaboration in oss projects,” in Collaborative Software Engineering. Springer, 2010, pp. 265–284.

[143] A. Jermakovics, A. Sillitti, and G. Succi, “Mining and visualizing developer networks from version control systems,” in Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering. ACM, 2011, pp. 24–31.

[144] B. Caglayan, A. B. Bener, and A. Miranskyy, “Emergence of developer teams in the collaboration network,” in Cooperative and Human Aspects of Software Engineering (CHASE), 2013 6th International Workshop on. IEEE, 2013, pp. 33–40.

[145] S. Panichella, G. Canfora, M. Di Penta, and R. Oliveto, “How the evolution of emerging collaborations relates to code changes: An empirical study,” in 22nd International Conference on Program Comprehension (ICPC). IEEE, 2014.

[146] J. Robertsa, I.-H. Hann, and S. Slaughter, “Communication networks in an open source software project,” in Open Source Systems. Springer, 2006, pp. 297–306.

[147] D. Damian, L. Izquierdo, J. Singer, and I. Kwan, “Awareness in the wild: Why communication breakdowns occur,” in Global Software Engineering, 2007. ICGSE 2007. Second IEEE International Conference on. IEEE, 2007, pp. 81–90.

[148] 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 (KCSD06). Citeseer, 2006, pp. 35–41.

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

[150] P. J. Adams, A. Capiluppi, and C. Boldyreff, “Coordination and productivity issues in free software: The role of Brooks’ law,” in Software Maintenance, 2009. ICSM 2009. IEEE International Conference on. IEEE, 2009, pp. 319–328.

[151] N. Nagappan, B. Murphy, and V. Basili, “The influence of organizational structure on software quality: an empirical case study,” in Proceedings of the 30th international conference on Software engineering. ACM, 2008, pp. 521–530.

[152] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, “Social coding in GitHub: transparency and collaboration in an open software repository,” in Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work. ACM, 2012, pp. 1277–1286.

[153] J. D. Herbsleb and R. E. Grinter, “Architectures, coordination, and distance: Conway’s law and beyond,” IEEE software, vol. 16, no. 5, 1999, pp. 63–70.

[154] M. Cataldo and J. D. Herbsleb, “Coordination breakdowns and their impact on development productivity and software failures,” Software Engineering, IEEE Transactions on, vol. 39, no. 3, 2013, pp. 343–360.

[155] K. Luther, K. Caine, K. Ziegler, and A. Bruckman, “Why it works (when it works): Success factors in online creative collaboration,” in Proceedings of the 16th ACM international conference on Supporting group work. ACM, 2010, pp. 1–10.

[156] K. Nakakoji, K. Yamada, and E. Giaccardi, “Understanding the nature of collaboration in open-source software development,” in Software Engineering Conference, 2005. APSEC’05. 12th Asia-Pacific. IEEE, 2005, pp. 8–pp.

[157] M. Foucault, J.-R. Falleri, and X. Blanc, “Code ownership in open-source software,” in Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering. ACM, 2014, p. 39.

[158] D. Posnett, R. D’Souza, P. Devanbu, and V. Filkov, “Dual ecological measures of focus in software development,” in 35th International Conference on Software Engineering (ICSE). IEEE, 2013, pp. 452–461.

[159] F. P. Brooks, Jr., The Mythical Man-month (Anniversary Ed.). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[160] B. S. Kuipers and M. C. De Witte, “Teamwork: a case study on development and performance,” The International Journal of Human Resource Management, vol. 16, no. 2, 2005, pp. 185–201.

[161] N. B. Moe, T. Dingsøyr, and T. Dybå, “A teamwork model for understanding an agile team: A case study of a scrum project,” Information and Software Technology, vol. 52, no. 5, 2010, pp. 480–491.

[162] M. Goeminne, M. Claes, and T. Mens, “A historical dataset for the gnome ecosystem,” in Proceedings of the 10th Working Conference on Mining Software Repositories. IEEE Press, 2013, pp. 225–228.

[163] A. Mockus, “Succession: Measuring transfer of code and developer productivity,” in Proceedings of the 31st International Conference on Software Engineering. IEEE Computer Society, 2009, pp. 67–77.

[164] C. Gutwin, R. Penner, and K. Schneider, “Group awareness in distributed software development,” in Proceedings of the 2004 ACM conference on Computer supported cooperative work. ACM, 2004, pp. 72–81.

[165] M. Gharehyazie, D. Posnett, and V. Filkov, “Social activities rival patch submission for prediction of developer initiation in oss projects,” in Software Maintenance (ICSM), 2013 29th IEEE International Conference on. IEEE, 2013, pp. 340–349.

[166] Y. Baruch, “Response rate in academic studies-a comparative analysis,” Human relations, vol. 52, no. 4, 1999, pp. 421–438.

[167] N. Blüthgen, F. Menzel, and N. Blüthgen, “Measuring specialization in species interaction networks,” BMC ecology, vol. 6, no. 1, 2006, p. 9.

[168] B. Vasilescu, A. Serebrenik, and M. van den Brand, “You can’t control the unfamiliar: A study on the relations between aggregation techniques for software metrics,” in Software Maintenance (ICSM), 2011 27th IEEE International Conference on. IEEE, 2011, pp. 313–322.

[169] A. Serebrenik and M. van den Brand, “Theil index for aggregation of software metrics values,” in Software Maintenance (ICSM), 2010 IEEE International Conference on. IEEE, 2010, pp. 1–9.

[170] W. Maalej and H.-J. Happel, “From work to word: How do software developers describe their work?” in Mining Software Repositories, 2009. MSR’09. 6th IEEE International Working Conference on. IEEE, 2009, pp. 121–130.

[171] ——, “Can development work describe itself?” in Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on. IEEE, 2010, pp. 191–200.

[172] P. Kampstra et al., “Beanplot: A boxplot alternative for visual comparison of distributions,” Journal of Statistical Software, vol. 28, no. 1, 2008, pp. 1–9.

[173] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, “Do code clones matter?” in Proceedings of the 31st International Conference on Software Engineering, ser. ICSE ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 485–495. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070547

[174] S. E. Sim, C. L. Clarke, and R. C. Holt, “Archetypal source code searches: A survey of software developers and maintainers,” in Program Comprehension, 1998. IWPC’98. Proceedings., 6th International Workshop on. IEEE, 1998, pp. 180–187.

[175] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes, “Sourcerer: a search engine for open source code supporting structure-based search,” in Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications. ACM, 2006, pp. 681–682.

[176] S. P. Reiss, “Semantics-based code search,” in Proceedings of the 31st International Conference on Software Engineering. IEEE Computer Society, 2009, pp. 243–253.

[177] S. Thummalapenta and T. Xie, “Parseweb: a programmer assistant for reusing open source code on the web,” in Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. ACM, 2007, pp. 204–213.

[178] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, “Mining stackoverflow to turn the ide into a self-confident programming prompter,” in Proceedings of the 11th Working Conference on Mining Software Repositories. ACM, 2014, pp. 102–111.

[179] “Github gist : https://gist.github.com/,” https://gist.github.com/.

[180] “pastebin: http://pastebin.com/,” http://pastebin.com/.

[181] “codeshare: https://codeshare.io/.”

[182] B. Ray and M. Kim, “A case study of cross-system porting in forked projects,” in Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. ACM, 2012, p. 53.

[183] B. Ray, M. Nagappan, C. Bird, N. Nagappan, and T. Zimmermann, “The uniqueness of changes: Characteristics and applications,” Microsoft Research Technical Report, Tech. Rep., 2014.

[184] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan, “A study of repetitiveness of code changes in software evolution,” in Proceedings of the 28th International Conference on Automated Software Engineering, ser. ASE, 2013.

[185] M. Kim, L. Bergman, T. Lau, and D. Notkin, “An ethnographic study of copy and paste programming practices in oopl,” in Empirical Software Engineering, 2004. ISESE’04. Proceedings. 2004 International Symposium on. IEEE, 2004, pp. 83–92.

[186] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer, “Genprog: A generic method for automatic software repair,” Software Engineering, IEEE Transactions on, vol. 38, no. 1, 2012, pp. 54–72.

[187] N. Meng, M. Kim, and K. S. McKinley, “Systematic editing: generating program transformations from an example,” in ACM SIGPLAN Notices, vol. 46, no. 6. ACM, 2011, pp. 329–342.

[188] ——, “Lase: locating and applying systematic edits by learning from examples,” in Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, 2013, pp. 502–511.

[189] R. Al-Ekram, C. Kapser, R. Holt, and M. Godfrey, “Cloning by accident: an empirical study of source code cloning across software systems,” in Empirical Software Engineering, 2005. 2005 International Symposium on. IEEE, 2005, pp. 10–pp.

[190] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in ACM SIGSOFT Software Engineering Notes, vol. 30, no. 5. ACM, 2005, pp. 187–196.

[191] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," Software Engineering, IEEE Transactions on, vol. 28, no. 7, 2002, pp. 654–670.

[192] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in Proceedings of the 29th international conference on Software Engineering. IEEE Computer Society, 2007, pp. 96–105.

[193] B. Vasilescu, K. Blincoe, Q. Xuan, C. Casalnuovo, D. Damian, P. Devanbu, and V. Filkov, "The sky is not the limit: Multitasking on GitHub projects," in International Conference on Software Engineering, ser. ICSE, 2016, to appear.

[194] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," Information and Software Technology, vol. 55, no. 7, 2013, pp. 1165–1199.

[195] M. Gabel and Z. Su, "A study of the uniqueness of source code," in Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. ACM, 2010, pp. 147–156.

[196] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2014, pp. 306–317.

[197] J. Ossher, H. Sajnani, and C. Lopes, "File cloning in open source java projects: The good, the bad, and the ugly," in Software Maintenance (ICSM), 2011 27th IEEE International Conference on. IEEE, 2011, pp. 283–292.

[198] G. Gousios, "The ghtorent dataset and tool suite," in Proceedings of the 10th Working Conference on Mining Software Repositories. IEEE Press, 2013, pp. 233–236.

[199] V. Bogdan, D. Posnett, B. Ray, M. v. d. Brand, Filkov, A. Serebrenik, D. Premkumar, and V. Filkov, "Gender and tenure diversity in github teams," ser. CHI '15. ACM, 2015.

[200] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in github," in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2014, pp. 155–165.