Elsevier Editorial System(tm) for Information and Software Technology
Manuscript Draft

Manuscript Number:

Title: A Systematic Mapping Study of Software Development with GitHub

Article Type: Review article

Corresponding Author: Dr. Valerio Cosentino,

Corresponding Author's Institution: Universitat Oberta de Catalunya

First Author: Valerio Cosentino

Order of Authors: Valerio Cosentino; Javier Luis Canovas Izquierdo; Jordi Cabot

Abstract: Context: GitHub, nowadays the most popular social coding platform, has become the reference for mining Open Source repositories, a growing research trend aiming at learning from previous software projects to improve the development of new ones. In the last years, a considerable amount of research papers have been published reporting findings based on data mined from GitHub. As the community continues to deepen in its understanding of software engineering thanks to the analysis performed on this platform, we believe it is worthwhile to reflect on how research papers have addressed the task of mining GitHub and what findings they have reported.

Objective: The main objective of our work is to identify the quantity, topic and empirical methods of research works targeting the analysis of how software development practices are influenced by the use of a distributed social coding platform like GitHub. %Other additional aspects can also provide further insights on the current state of the art, like methodologies and technologies used to conduct research in GitHub, or community analysis and publication fora.

Method: A systematic mapping study was conducted with four research questions and assessed 80 publications from 2009 to 2016.

Results: Most works focus on the interaction around coding-related tasks and project communities. We also identify some concerns about how reliable are these results based on the fact that, overall, papers use small datasets, poor sampling techniques, employ a scarce variety of methodologies and/or are hard to replicate.

Conclusions: Our study attested the high activity of research work in the field, revealed a set of shortcomings and proposed some actions to mitigate them.

A Systematic Mapping Study of Software Development with GitHub

Valerio Cosentino[a,*], Javier L. Cánovas Izquierdo[a], Jordi Cabot[b,]

[a] *UOC, Barcelona, Spain*
[b] *ICREA, Barcelona, Spain*

Abstract

Context: GitHub, nowadays the most popular social coding platform, has become the reference for mining Open Source repositories, a growing research trend aiming at learning from previous software projects to improve the development of new ones. In the last years, a considerable amount of research papers have been published reporting findings based on data mined from GitHub. As the community continues to deepen in its understanding of software engineering thanks to the analysis performed on this platform, we believe it is worthwhile to reflect on how research papers have addressed the task of mining GitHub and what findings they have reported.

Objective: The main objective of our work is to identify the quantity, topic and empirical methods of research works targeting the analysis of how software development practices are influenced by the use of a distributed social coding platform like GitHub.

Method: A systematic mapping study was conducted with four research questions and assessed 80 publications from 2009 to 2016.

Results: Most works focus on the interaction around coding-related tasks and project communities. We also identify some concerns about how reliable are these results based on the fact that, overall, papers use small datasets, poor sampling techniques, employ a scarce variety of methodologies and/or are hard to replicate.

Conclusions: Our study attested the high activity of research work in the field, revealed a set of shortcomings and proposed some actions to mitigate them.

*Corresponding author
Email addresses: vcosentino@uoc.edu (Valerio Cosentino), jcanovasi@uoc.edu (Javier L. Cánovas Izquierdo), jordi.cabot@icrea.cat (Jordi Cabot)

Keywords: GitHub, Open Source, Mining software repositories, Systematic mapping study

1. Introduction

Software forges are web-based collaborative platforms providing tools to ease distributed development, especially useful for Open Source Software (OSS) development. GitHub represents the newest generation of software forges, since it combines the traditional capabilities offered by such systems (e.g., free hosting capabilities or version control system) with social features [1]. In recent years, the platform has witnessed an increasing popularity, in fact, after its launch in 2008, the number of hosted projects and users have grown exponentially[1], passing from 135,000 projects in 2009 to more than 35 million projects in 2015.

At the heart of GitHub is Git [2], a decentralized version control system that manages and stores revisions of projects based on master-less peer-to-peer replication where any replica of a given project can send or receive any information to or from any other replica. Despite the close relation with Git, GitHub comes with many of its own features specially aimed at facilitating the collaboration and social interactions around projects (e.g., issue-tracker, pull request support, watching and following mechanisms, etc.). Additionally, the platform provides access to its hosted projects' metadata, available through the GitHub API[2], thus facilitating further analysis.

Such social features, the open API plus its popularity among the software developers community make GitHub the best candidate to provide the raw data required to analyze and better understand the dynamics behind (OSS) development communities as well as the impact of social features in (distributed) software development practices [3, 4]. Therefore, more and more Software Engineering (SE) researchers have turned to GitHub as the center of their empirical studies.

In this paper, we present a systematic mapping study of all papers reporting findings that rely on the analysis and mining of software repositories in GitHub. After

[1] http://redmonk.com/dberkholz/2013/01/21/github-will-hit-5-million-users-within-a-year/

[2] https://developer.github.com/v3/

a systematic selection and screening procedure, we ended up with a set of 80 papers that were carefully studied. Our goal was to study how software development practices have changed due to the popularization of social coding platforms like GitHub and how projects could optimize their collaboration and management schemas to optimize them. A secondary goal was to analyze different quality aspects of the empirical methods employed by the research works themselves in order to assess the representativeness and reliability of those findings. As far as we know this is the first work conducting a meta-analysis of papers reporting results based on GitHub data mining.

This paper is organized as follows: Section 2 describes the procedure we followed in this study. Section 3 presents the research questions, Section 4 describes the data acquisition process and Section 5 reports on the analysis and results obtained. Section 6 presents some discussion about our findings and Section 7 comments on the threats to validity. Finally, Section 8 concludes the paper and presents some further work.

2. Methodology

To perform our mapping study we followed a procedure inspired by previous works [5, 6]. The procedure has five phases, shown in Figure 1, namely: (1) definition of research questions; (2) conduct search, where primary studies are identified by using search strings on scientific databases or browsing manually through relevant venues; (3) screening of papers, where inclusion/exclusion criteria are applied to remove those works not relevant to answer the defined research questions; (4) classification scheme, where dimensions to classify each paper according to each research question are identified; and (5) data extraction and mapping study, where the actual data extraction takes place.

We have organized these phases into three main groups (see gray boxes in Figure 1): (1) research questions, (2) data acquisition, where we describe how we selected and screened papers (i.e., second and third phases); and (3) analysis and results, where we report on the classification employed and the resulting systematic mapping (i.e., the last two phases). This structure allows us to report on both the classification scheme and the results for each research question in the same section, which we believe improves

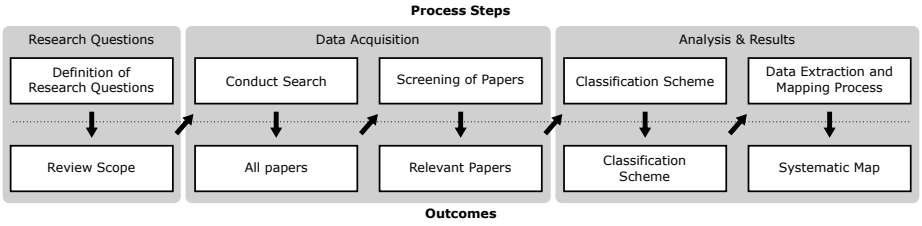


Figure 1: Systematic mapping study process [5].

the reading flow. Next sections describe each one of these phase groups.

3. Research Questions

The main goal underlying our work is to provide an overview of research efforts focusing on the analysis of all kinds of software development practices relying on / influenced by the use of the GitHub platform as the most representative example of the current generation of software forges. This overview must be accompanied with a characterization of the empirical methods used in those analysis, the tools employed in them and the research community behind those works. Therefore, our mapping study addresses the following research questions:

RQ1: What topics/areas have been addressed?
Rationale: Our interest is finding out the main areas (and key topics in those areas) that have been targeted while studying how GitHub has influenced software development.

RQ2: What empirical methods have been used?
Rationale: Our intent here is to analyze and classify the methods commonly applied in research works around GitHub.

RQ3: What technologies have been used to extract and build datasets from GitHub?
Rationale: The study of GitHub generally implies first the selection and retrieval of a subset of GitHub repositories. Our goal is to catalogue the main technologies used for that purpose.

RQ4: What is the research community behind these works like?

Digital Library	URL	Adv.Query	Cit. Nav.	Ref. Nav.
Google Scholar	scholar.google.com	-	✓	-
DBLP	dblp.uni-trier.de	✓	-	-
ACM	dl.acm.org	✓	✓	✓
IEEE Xplore	ieeexplore.ieee.org	✓	-	✓
ScienceDirect	sciencedirect.com	✓	-	-
CiteSeerX	citeseerx.ist.psu.edu	✓	✓	✓

Table 1: Digital libraries selected and main features provided.

Rationale: GitHub papers are published in multiple and different venues by a sizeable set of researchers. Our goal is to characterize this research subcommunity and the publication venues they focus on. This can help to interpret better the results of previous questions, to encourage new collaborations among isolated subgroups and to help identifying potential venues to publish new results in this area.

4. Data Acquisition

This section describes the search process and the selection criteria for our mapping study.

4.1. Conduct search for primary sources

The goal of this step is to build an exhaustive collection of works in the field of study. We defined a three-phased collection process in order to make the set of evaluated works as complete as possible.

The first phase consisted in selecting a set of digital libraries, defining the corresponding search queries and finally executing those. The selection of the digital libraries was driven by several factors, specifically: (1) number of works indexed, (2) update frequency, and (3) facilities to execute advanced queries and navigate the citation and reference networks of the retrieved papers. We selected 6 digital libraries (see Table 1) that presented a good mix of the desired factors.

Digital Library	Search query
IEEE Xplore	`((((("Publication Title":github) OR "Abstract":github) OR "Index Terms":github) OR "Author Keywords":github)`
DBLP	Search by `title`
ACM	`acmdlTitle:(github) AND recordAbstract:(+github) AND keywords.author.keyword:(+github)`
Google Scholar	`allintitle:github`, selecting hits with a navigable link and in English
ScienceDirect	`TITLE-ABSTR-KEY(github)[All Sources(Computer Science)]`
CiteSeerX	Three different searches: `title:github`, `abstract:github` and `keyword:github`

Table 2: Executed queries for each digital library.

Once the digital libraries were determined, search queries were defined to retrieve the initial selection of works to be filtered and screened later on. Table 2 shows the queries we defined for each digital library. In general, we searched for all works that contained the word *GitHub* (or variations of it, i.e., *github*, *git hub*) in either the title, abstract, author keywords or index terms. This resulted in a collection of 488 works (removing duplicates). To refine the search, a title/abstract pruning process was applied to discard those works which clearly were not studying GitHub itself (e.g., papers just mentioning GitHub as the platform where they were making publicly available their artifacts or research data) and therefore out of the scope of our study. The pruning process removed 173 papers. Thus, we ended up this first phase with 315 works.

The second phase took the previous set and performed a breadth-first search using backward and forward snowball methods by navigating their citations and references. To this aim, citation and reference networks provided by the digital libraries were used (when possible, otherwise we followed a manual approach). Each new work found was then used in turn to identify subsequent new ones. We collected 65 additional papers as a result of this phase, to which we also applied a title/abstract pruning process. As a result, only 21 new papers were finally added to the collection due to snowballing. At the end of this second phase, our collection contained 336 works.

In the third phase, an edition-by-edition (or issue-by-issue) manual browsing of the

main conference proceedings and journals was performed. The goal of this phase is to complete the list of the initial works and also assess the completeness of the selection obtained so far. We selected 24 venues in total (16 conferences and 8 journals, shown in Table 3) from January 2009 until July 2016. This phase identified 6 new works. Again, we applied a title/abstract pruning process but no work was discarded so the 6 works were added to our collection.

At the end of the search process we obtained a total of 342 works, 92.10% coming from the first phase, 6.14% from the second one and 1.76% from the third phase. The upper part of Table 4 summarizes the number of works initially collected, pruned and eventually selected in each phase of this process.

4.2. Selection Criteria and Screening

Next, we applied a two-phased selection process in order to determine which works were considered relevant to answer our research questions.

First, the collected works were classified into three dimensions according to how they used GitHub.

- Papers using GitHub simply as a "project warehouse" to draw repositories from were classified as *source*, as they relied on GitHub just as a source of repositories and did not benefit or study any specific GitHub feature (i.e. the paper would be the same if instead of GitHub projects had been stored in file folders in a server).
- Papers analyzing how software projects were using GitHub features were classified as *target*, as the GitHub platform itself was the main object of study.
- Works studying GitHub as part of a comparative studies on open-source development environments and platforms (GitHub was studied but only in relationship to other platforms and therefore it was not the main object of study) were classified as *description*.

In total we found 167 *source* papers, 157 *target* papers and 18 *description* papers. In our study we are interested in those works classified as *target*, as they are the ones that aim to provide insights on how the platform and its characteristics are being used in software development.

Conferences	
CSCW	Computer-supported cooperative work
CSMR	European Conference on Software Maintenance and Reengineering
MSR	International Conference on Mining Software Repositories
ICSM(E)	International Conference on Software Maintenance and Evolution
ICSE	International Conference on Software Engineering
FSE	International Symposium on the Foundations of Software Engineering
ISSRE	International Symposium on Software Reliability Engineering
APSEC	Asia-Pacific Software Engineering Conference
SANER	International Conference on Software Analysis, Evolution, and Reengineering
WCRE	Working Conference on Reverse Engineering
ESEM	Empirical Software Engineering and Measurement
SEKE	International Conference on Software Engineering and Knowledge Engineering
WWW	International Conference on World Wide Web
OSS	International Conference on Open Source Systems
SAC	Symposium on Applied Computing
EASE	International Conference on Evaluation and Assessment in Software Engineering
Journals	
TOSEM	Transactions on Software Engineering and Methodology
TSE	Transactions on Software Engineering
SoSym	Journal on Software & System Modeling
Software	IEEE Software
JSS	Journal of Systems and Software
ESE	Empirical Software Engineering
IST	Information and Software Technology
SCP	Science of Computer Programming

Table 3: Selected venues.

These *target* papers were the input of the subsequent screening process. At this point, we discarded the following groups of papers as we considered irrelevant for this particular study:

- Papers studying the use of GitHub in non-software development projects (e.g.,

Search for primary sources				
Phase	Description	Initial Set	Pruning	Final Set
1	Digital Libraries	488	173	315
2	Snowballing	65	44	21
3	Other Venues	6	0	6
			Total	342

Selection and screening Process			
Phase	Description	Category	Final Set
1	Classification	Source	167
		Target	157
		Description	18
		Total	342
2	Screening	**Selected**	**80**
		Discarded	77
		Total	157

Table 4: Number of works considered in the search and selection/screening processes.

education [7], collaboration on text documents [8, 9], open government practices [10], for replicability of scientific results [11]).

- Papers analyzing niche technologies (e.g., development and distribution of R packages [12], adoption of database frameworks in Java projects [13]) due to the challenge of generalizing those results to any kind of software project.
- Papers subsumed by another paper from the same authors presenting more complete results. In those cases we only kept the extended version.
- Papers not written in English or not published in peer-reviewed conferences or journals (e.g. technical reports, masters and PhD thesis).
- Papers that rely on GitHub to evaluate new algorithms or tools (e.g., recommenders [14, 15], predictors [16, 17]) proposed in the paper, even if that algorithm is related to a GitHub feature, since the paper uses GitHub more of as a validation platform.

After running the screening process according to this criteria, we retained 80 and discarded 77 of the papers[3]. The bottom part of Table 4 shows the number of works classified and screened during these phases.

5. Analysis and Results

5.1. RQ1: What topics/areas have been addressed?

In this section we summarize the main findings reported by the selected works, grouped in topics and areas of research interest to get a better overview of what the contributions of those papers are and what we can learn from them.

To classify the papers we applied a grounded theory approach to analyze those findings. First, each paper was labeled with an identifier and analyzed to summarize its main findings . We then performed open coding on such findings with the purpose of assigning them to specific topics. Finally, we grouped the topics conceptually similar, resulting into four main areas of research, focused on: (1) development, (2) projects, (3) users and (4) the GitHub ecosystem itself.

Topic identification was based on the common terminology employed around the field of software forges, with special attention to specific vocabulary linked to GitHub features and development model. As a web-based hosting service for collaborative development projects using the Git control system, projects and users can be considered the main assets of the platform. Each project keeps track of the submitted issues, pull requests and commits, which are therefore other potential topics of study. Pull requests are the main means to contribute to a project. To create a pull request, the user has first to fork a project, and once her changes have been completed send the pull request to the original project asking for those changes to be integrated (i.e. "pulled") in the project. GitHub also supports social features like followers and watchers which have been also the topic of study for some research works.

Tab. 5 summarizes the paper distribution along the 4 areas of research interest and the corresponding topics. Note that some works may report findings for different top-

[3]The full list of collected and selected/discarded papers is available at http://tinyurl.com/systmap-github-download

Areas	Topics	In topic	In area
Software development	Code contributions	24	37
	Issues	6	
	Forking	12	
Projects	Characterization	21	45
	Popularity	14	
	Communities & teams	17	
	Global discussions	12	
Users	Characterization	21	34
	Rockstars	10	
	Issue reporters and assignees	4	
	Followers	10	
	Watchers	6	
Ecosystem	Characterization	9	24
	Transparency	8	
	Relationship with ther platforms	7	

Table 5: Distribution of works across the areas of interest and topics.

ics and areas at the same time and therefore the total number is higher than the size of our paper collection set. The software development area contains findings about code contributions, issues and forking. The project area includes findings about the different types of project in GitHub, their popularity, communities and teams in them and the characterization of the discussions that may arise in the community during the development. The third area reports findings about different types of users, such as popular users (i.e., rockstars), issue reporters and assignees, watchers and followers. The last area contains findings about the GitHub ecosystem, such as its characterization, transparency, and the relations with other platforms (e.g., Twitter, StackOverflow).

With regard to areas, findings concerning projects (45) and software development (37) overtake those about users (34) and platform (24). On the topic side, the ones that have received more attention concern code contributions (24) and general findings on

projects (21). Next sections reports of the findings according to the areas of interests and topics identified.

5.1.1. Software development - Code contributions

Contributions on the platform are unevenly distributed. Most code contributions are highly skewed towards a small subset of projects [18, 19], exhibiting a power-law distribution.

Few developers are responsible for a large set of code contributions. Avelino et al. report that most projects have a low truck factor[4] (i.e., between 1 and 2), meaning that a small group of developers is responsible for a large set of code contributions [20]. A similar finding is reported also by other works, Kalliamvakou et al. claim that more than two thirds of projects have only one committer (the project's owner) [21]. Pinto et al. note that a large group of contributors is responsible for a long tail of small contributions [22], while Yamashita et al. acknowledge that the proportion of code contributions does not follow the Pareto principle[5], on the contrary, a larger number of contributions is made by few developers [23].

Most contributions come in the form of direct code modifications. A code contribution can be made either directly (pushed commits) or indirectly (pull requests). The former are exclusively performed by developers granted write permission on the project, while the latter can come from anyone. Most code contributions are performed by direct code modifications [24].

Pull requests' characterization. Most pull requests are less than 20 lines long, processed (merged or discarded) in less than 1 day and the discussion spans on average to 3 comments [24]. Furthermore, they are evaluated using manual and automatic testing

[4]The truck factor is a measurement of the concentration of information in individual team members. It connotes the number of team members that can be unexpectedly lost from a project before the project collapses due to lack of knowledgeable or competent personnel

[5]The Pareto principle or the so-called "80-20 rule" states that 80% of the contributions are performed by roughly 20% of the contributors

as well as code reviews [25].

Importance of casual contributions. A specific type of pull request is defined as "drive-by" commits ([26]) or casual contributions ([22]). According to [24], casual contributions account for 7% of the pull-requests made to GitHub projects in 2012, and a more recent study ([22]) reports that casual contributors are far from being trivial and rather common in GitHub (48.98%) even if many of them are minor fixes.

Technical factors to accept pull requests. Some works have identified technical factors that influence on the acceptance (or rejection) of an indirect code contribution. Pull requests fully addressing the issue they are trying to solve, self contained and well documented [25] as well as including test cases [27, 28] and efficiently implemented [29] are more likely to be accepted, since they come with enough means to ease their comprehension and to assess their quality. In addition, pull requests targeting relevant project areas or recently modified code [24, 30], fixing known project bugs, updating only the documentation [31] or easily verifiable [32] have also higher probability to get accepted, since they have higher priority or are simpler to check.

Technical factors to reject pull requests. On the contrary, pull requests containing unconventional code (i.e., code not adhering to the project style) [24, 33], suggesting a large change [34], introducing a new feature, or conflicting with other existing functionality [32] are less likely to go through.

Social factors to accept pull requests. Social factors also influence the acceptance of pull requests [27, 28, 35, 34]. Contributions from submitters with prior connections to core members, having stronger social connection (e.g., number of followers) or holding a higher status in the project are more likely to be accepted [27, 28, 35]. In particular, Soares et al. report that contributions made by members of the main team increase in 35% the probability of merge and when a developer has already submitted a pull request before, his or her contribution has 9% more chance of being merged [34].

Social factors to reject pull requests. A pull request from an external collaborators have 13% less chance of being accepted, and if it is the first contributions to the project,

the chance of merge is decreased in 32% [34]. Also, contributions sent to popular, well-established (i.e., mature) projects or with a high amount of discussion are less likely to be accepted [27, 33].

Geographical factors to accept pull requests. When submitters and integrators are from the same geographical location there is 19% more chances that the pull requests will get accepted [36]. This is due to the fact that in general integrators (i) perceive that it easy to work with submitters from the same geographical location and (ii) encourage submitters from their geographical location to participate [36].

Pull request evaluation time. The latency of processing pull requests has been discussed in different works. Complex and large pull requests, arising long discussions, undergoing code review processes, touching key parts of the system or having low priority are associated with longer evaluation latencies [37, 24, 30]. Also pull requests from unknown developers undergo a more thorough assessment, while contributions from trusted developers are merged right away [26]. Note that the increase in the time needed to analyze a pull request reduces the chances of its acceptance [34].

Moreover, Yu et al. reports also that a minor impact on the evaluation time is due to pull requests submitted outside "business hours" and on Friday and missing links to the issue reports [37]. On the other hand, pull requests submitted by core team members, contributors with more followers, more social connections within the project, and higher previous pull request success rates are associated with shorter evaluation latencies. Also the use of @-mention (to ping other developers), code reviews help reducing the evaluation time. In particular, two works report that even if @-mention is not widely used in pull requests, pull requests using @-mention tags tend to be processed quicker [37, 38].

5.1.2. Software development - Issues

Issues tracker are scarcely used. Even if issue trackers are only explicitly disabled in 3.8% of the projects, 66% of the projects do not actually use them [39].

Few projects attract most of the issues. Issues are unevenly distributed on the projects

that actively use the GitHub issue tracker. In particular, the distribution of open issues follows a power-law distribution [40]. Issues are generally sent to popular projects, with large code bases and communities [39].

Distribution of issues in a project. The number of opened issues is on average higher right after the project creation, while it tends to decrease few months later. However, the number of opened issues is stable or exhibits slightly negative trend over time. Conversely, pending issues are constantly growing [41].

Small number of labels are used to tag issues. GitHub offers a label mechanism to ease the categorization of issues, however less than 30% of issues are tagged [39, 42]. The vast majority of projects that use labels, only use 1 or 2 different ones (45.53% and 25.42% respectively) [42]. The most common ones are *bug*, *feature* and *documentation* [39, 42]. *Bug* and *enhancement* are often used together [42].

Issue lifetime. Issue lifetimes are generally stable in a project [41]. In particular, the response time to adress issues is generally fast. However, the older an issue is, the smaller is the chance that it will be addressed. [43]. It is important to note that the size of the community behind a project does not seem to reduce the time to handle issues [39, 41]. Finally, the number of projects/developers followed by team members has a positive effect on fixing long-term bugs, not on 'rapid response to user issues [43].

The use of labels and @-mentions is effective in solving issues. The use of labels [42] and @-mentions [44] has a positive impact on the issue evolution by enlarging the visibility of issues and facilitating the developers collaboration, and eventually leading to an increase in the number of issues solved. However, labeled and @-mentioned issues are likely to need more time to deal with than other issues [42, 44].

5.1.3. Software development - Forking

Forking is unevenly distributed. Most of the projects in GitHub are never forked [29, 18, 21, 45]. In particular, the distribution of forks follow a power-law distribution [40, 46], meaning that there are lots of projects with few forks and few projects forked

a very large number of times [18].

Forking as good indicator of project liveliness. The number of forks of a project is positively correlated with the number of open issues, watchers [40] and stars [47], as well as with number of commits and branches [43], thus reflecting that active projects are often more popular. However, it is important to note the number of forks does not correlate with the number of pull requests accepted [48].

Forks are mostly used to fix bugs. There exist 3 kinds of forks in GitHub according to Rastogi and Nagappan. In particular, *Contributing forks* are those forks aimed to end up as pull requests to the forked project to integrate changes, *independently developed forks* are forks that do not send pull requests to the forked project but have internal commits probably deviating from the mother project; and *inactive forks* do neither send nor receive pull requests nor have internal commits [49]. The last two categories are the common ones, since most forks do not retrieve new updates from original projects [50]. Instead, the contributing forks can be further classified. In particular, forks are mostly used to fix bugs and add new features [46, 50], and less frequently to add documentation [46].

Forking is beneficial. Forking is considered positive for several reasons such as to preserve abandoned programs, to improve the quality of a project (e.g., adding testing and debugging), give developers control over the code base (e.g., experiments and customization of existing projects) [50] as well as submitting pull requests and making contributions to the original repositories [46]. Moreover, even if forking could also have some drawbacks, such as confusion (e.g., many instances of the same project), fragmentation (e.g., development efforts over from multiple project versions, bug-fixes not propagated) and compatibility between the forked and original projects [50], it seems this is not often the case as reported by Jiang et al. (i.e., forking does not split developers into different competing and incompatible versions of repositories) [46].

Forkability. The chances to get a project forked depends on different factors. According to [51], project where developers provide additional contact information publicly

(emails and personal web site's urls), that are clearly active or with popular project's owners [46] are more probable to be forked. Prior social connections between the project owner and the forker, an increase in developer community size for medium-size projects [49] and projects written in the forker preferred programming language can increase the chances of forking [46].

5.1.4. Projects - Characterization

Most projects are personal. Most projects are personal and little more than code dumps such as example code, experimentation, backups or exercises not intended for customer consumption [29, 52, 45, 21]. Thus, they exhibit very low activity (e.g., commits) and attract low interest [45, 21] (e.g., watchers and downloads [53]).

Most projects choose not to benefit from GitHub features. Most projects ignore GitHub collaboration capabilities [45]. In particular, the pull request mechanism is rarely employed [29, 21] (according to the authors in [24], 14% of projects used it) as well as the use of the issue tracker, fork and issue label mechanisms [40, 42]. Sometimes, these project rely on external tools to manage the collaboration [21].

Commercial projects on GitHub. GitHub is not only for open source software. Commercial projects also use GitHub and the functionalities provided by the platform. In particular, they adopt a workflow that builds on branching and pull requests to drive independent work. Pull requests are also used to isolate individual development and perform code review before merging as well as to act as coordination mechanism. Coordination is also achieved using GitHub's transparency and visibility, while conflict resolution is often based on self-organization when assigning tasks [54].

Project domains. GitHub projects can be classified according to the type of software they aim to develop. Popular ones are *Application software* offering software functionalities to end-users, like browsers and text editors; *system software*, which provides services and infrastructure to other systems, like operating systems, middleware, servers, and databases; *software tools*, which provide support to software development tasks, like IDEs, package managers, and compilers; *documentation* used to host documenta-

tion, tutorials, source code examples and *web* and *non-web* libraries and frameworks. The top three domains are system software, web and non-web libraries, followed by software tools [47].

Project languages. JavaScript, Ruby, Python, Objective-C and Java are the top used languages in terms of number of projects in GitHub[55, 56]. Furthermore, the top 5 projects in terms of the number of contributors are Linux kernel related, and two of which are owned by Google [57].

User base and documentation keep the project alive. Projects that are actively maintained are more likely to be alive in the future than projects that only show occasional commits [58], as well as a good interaction with the user base [58, 59]. The presence of documentation correlates as well with future project survival (in particular the projects that include *contributing.md* are 25% to 45% more likely to be alive) [58].

Attracting and retaining contributors. The ability to attract and retain contributors can influence the projects survival. Simple contribution processes [60], presence of documentation [61], project complexity and popularity [40] are generally associated with the ability to attract contributors. Conversely, projects used professionally [60], the adoption of more distributed and transparent (non-centralized) practices as pull requests are generally associated to projects with higher contributor retention [62, 59]. Projects are better at retaining contributors than at bringing on new ones [60].

5.1.5. Projects - Popularity

Few projects are popular. Only few projects are popular, in particular the distribution of stars and downloads ([43],[53]) (key metrics for popularity) follows a power-law distribution.

Benefit of project popularity. Popularity is useful to attract new developers [40, 51, 60], since the project is perceived as a high quality and worthwhile.

Project activity and project popularity. For GitHub projects, good indicators of popularity are the number of forks, stars, followers and watchers [29, 19, 46]. The

number forks positively and strongly correlates with the number of stars and watchers [40, 47], as well as with the number of commits and branches [43], furthermore the number of stars grows exponentially with the number of contributors [43].

Documentation as a factor for project popularity. There exists a clear relationship between the popularity and the documentation effort [63, 64, 61]. In particular, popular projects exhibit higher and more consistent documentation activities and efforts [63], 95% popular projects have nonempty READMEs (and larger than unpopular counterparts) [64], setting up useful documents can be the key of attracting coding contributors [61]. Moreover, popular projects often specify the programming language used [43], rely on testing mechanisms [64], and have a wiki [53].

Fame of projects driven by organizations. Projects owned by organizations tend to be more popular than the ones owned by individuals [47].

Trends in programming languages and domain as factors for project popularity. The adoption of a given programming language and the application domain may also impact the project popularity [47, 59]. The most popular projects are the ones adopting Ruby, JavaScript, Java or Python [40]. However, JavaScript is responsible for more than one third of them [47], and together with Ruby, accounts for most of the projects in GitHub [65].

Project popularity vs. user popularity. According to Jarczyk et al., projects whose developers follow many others are in general more popular [43]. Conversely, this does not happen for projects whose developers are followed by many others (i.e., having popular developers in the project). Calvo-Villagran and Kukreti confirm this finding by reporting that the involvement of one or more popular users on a project does not influence on the projects popularity [65].

5.1.6. Projects - Communities & teams

Small development teams. Most of the projects have small development teams [53]. Kalliamvakou et al. claim that 72% of projects have only have a single contributor [21],

Lima et al. report that 74.22% of projects have two [18], while Avelino et al. report that a small group of developers is responsible for a large set of code contributions [20]. Similarly, the work in [52] states that the vast majority of projects in GitHub have fewer than 10 contributors, and the work in [23] affirms that 88%-98% of projects have fewer than 16.

Types of team. Vasilescu et al. report that 4 different kinds of communities exist in GitHub, called *fluid*, *commercial*, *academic* and *stable* respectively. Fluid communities are composed of voluntary developers that "come and go as their interest waxes and wanes". Commercial communities are made of professionals, thus changes in team composition are mostly due to the company's dynamics. Academic communities are made by people working in academia, thus such communities are less dynamics then the previous ones. Finally, stable communities do not exhibit at all changes in their composition, since they are mostly personal or small-team projects [66].

Teams and diversity. There exist a positive impact on the collaboration experience and productivity in diverse (gender, tenure, nationality) teams, despite some possible drawbacks such as improper contributions by newcomers and difficulties in communication due to different user origins [66, 67]. Location diversity is not a rare occurrence in GitHub [67], but is generally observed in teams with more than 40 users, while small teams tend to have developers concentrated in the same location [18, 40]. Tenure diversity is made possible by the lowered barrier to participation GitHub offers [67]. It may increase attrition between team members, however this negative effect appears to be mitigated when more experienced people are present) [67]. Finally, regarding gender diversity, only 1% of projects in GitHub are all-female projects [68, 67].

Community composition. The definition of who is part of the project community varies across projects. Many works have proposed classifications of the users involved to a project. Vasilescu et al. reports that "everyone who does something in the project" (e.g., pushes code, submits pull requests, reports issues) is considered part of the community [66], while Yamashita et al. identify two kinds of users, core and non-core developers (where the former are granted with write permission on the project while

the latter are not) [23]. Other works [69, 70, 71, 72] provide more detailed structures by relaying on the user experience, coding activity, popularity and actions on the platform (e.g., watching, forking, commenting, etc.).

Stability of core developers. The proportion of core developers remains stable as the project gets larger [71, 23].

Few users become core developers. Commonly, long-term contributors are turned into core developers, so that they can help developing big projects. However, this kind of collaboration is quite rare [18]. In addition, it may not be possible when the contributions are done to popular projects [73, 33] or projects that rely on paid developers [43, 61].

5.1.7. Projects - Global Discussions

Most discussions revolve around code-center issues. Discussions are engaged by core and non-core developers in order to evaluate both the problem identified and the solution proposed [28]. Discussion topics are mostly around the code, due to the code-centric view of the collaboration in GitHub.

Long discussions have a negative effect. Long discussions generally entail a negative effect on the final decision about the contribution [32, 27, 28, 37, 33]. However, they show that projects developers are willing to engage with users to clarify issues [74].

Mood in discussions. Mood in discussions is generally neutral due to their technical nature [75]. However, negative emotions can be identified in discussions on Monday, in projects developed in Java [75] [75] and in security-related discussions [76]. Finally, Country location diversity has a positive effect on discussions [66, 75].

Barriers and enablers in discussions. Paid developers can be a barrier to attract users for discussion [61], while generally discussions using label and @-mention mechanisms tend to involve more people in discussions [42, 28, 38].

5.1.8. Users - Characterization

A majority of users come from the United States. Several works [18, 77, 73, 31, 55] report that most of the users in GitHub come from North America, Europe and Asia. In particular, United States accounts for the largest share of the registered user accounts [77].

GitHub is mainly driven by males and young developers. Most of users in GitHub are males, while females account only for a small percentage of GitHub [55, 67]. In addition, most of the users are between 23-32, and only a tiny percentage is older than 60 [55]. A similar finding is reported by Vasilescu et al. where the median and mean of the users' age in their survey is 29 and 30 respectively [67]. Users in GitHub have more than 6 years of development experience [78, 67].

Hobbyists as main citizens. The large majority of users are hobbyists who make occasional contributions, while a tiny percentage are dedicated developers who produce a stream of contributions over long term [29] with peaks during the working hours in Europe and North America [79].

Users and private projects. Approximately half of GitHub's registered users work in private projects, thus their activities on the platform are not publicly visible [21].

Indicators to assess the expertise, interest and importance of users. The features provided by the platform offer hints to identify users' expertise, interests and their importance in the community, thus supporting more accurate impressions of contributors [80, 19]. In particular, the user expertise is often judged by the breadth and depth of the projects the user owned, and the coding languages she uses; while hints to her interests are often signaled by the feed of his actions across projects [29, 32, 81]. Furthermore, Badashian and Stroulia report that the number of times the repositories of a user have been forked is an indicator of the value of the content produced by the user [56]. This information is heavily used in hiring processes.

Commitment. Sustained contributions and volume of activity of a user are considered

as a signal of commitment to the project [29, 32, 81, 19]. The commitment of the users in a project may highly differ each other. For instance, Dabbish et al. report that 42% of issue reporters generally do not contribute to the code base of the target project [39], while Onoue et al. claim that some developers are balanced in doing coding, commenting, and issue handling, while others focus more on code or comments [82].

Productivity. User productivity depends on factors such as the number of projects the user is involved and her commitment to the project. In particular, a user that focuses on many projects may decrease the quality of his contributions to the single projects (e.g., increase the number of bugs in the code) [43, 83]. Furthermore, distractions and interruptions from communication channels negatively impact developer productivity as well as geographic, cultural, and economic factors can pose barriers to participation through social channels [55]. Conversely, prior social connections [35], proper onboarding support and an efficient and effective communication of testing culture [26] can increase the productivity of new contributors, that will face less pull request rejections.

5.1.9. Users - Rockstars

Who is a rockstar? A rockstar, or popular user, is characterized by having a large number of followers, which are interested in how she codes, what projects she is following or working on [29, 19, 56]. The number of followers a user has is interpreted as a signal of popularity/status in the community [29, 19]. Rockstars are loosely interconnected among them and tend to have connections with unpopular users [18, 84]. In addition, rockstars exert their influence in more than one programming languages [56].

How to become a rockstar? Users become popular as they write more code and monitor more projects. However, while low popularity levels can be attained with a little effort, achieving higher levels requires much more effort [85]. Popularity is not gained through development alone, in fact Lima et al. report that the relation between the number of followers of a user and her contributions is not strong, meaning that a higher level of activity does not directly translate into a larger number of followers

[18]. Thus, there are other factors that improve the user popularity such as participating in different projects and discussions [85].

Impact on the project. Rockstars play an important role on the project dissemination and attractiveness. In particular, when a rockstar increases her activity on a project, it attracts more followers to participate (e.g. opening/commenting issues) on the same project [86, 81, 87]. However, it is worth noting that the involvement of one or more popular users will not influence on the project's popularity [65, 56].

5.1.10. Users - Issue reporters and assignees

Activity. A vast majority of issue reporters are not active on code contributions, do not participate in any project and do not have followers [88], in addition they often have an empty profile [32]. On the other hand, a considerable number of assignees is very active, and in comparison with reporters, they are more popular, exhibit higher code activity and have been on GitHub for more years [88].

5.1.11. Users - Followers

Following is not mutual. The following relationships in GitHub are characterized by low reciprocity and follow a power-law distribution, meaning that given two uses, they do not follow each others back in general, and only few users have a high number of followers while the majority does not [81, 18, 89, 57, 53].

Why following other users? The mechanism of following allows users to receive notifications about what other people are working on and who they are connecting with. Following is an action that is used with different intents. It is used as an awareness mechanism, to discover new projects and trends, for learning, socializing and collaborating as well as for implicit coordination [90, 87, 46] and when looking for a job [40].

Followers are not special. Users who follow many others are not much more active than those who do not [18]. In the same direction, Alloho and Lee claim that there

exists a weak correlation between the number of users that follow many others and the number of project participations[40].

The *small world* of followers. The following networks in GitHub fit the theory of the "six degrees of separation" in the "small world" phenomenon, meaning that any two people are on average separated by six intermediate connections [84].

5.1.12. Users - Watchers

Why watching projects? Watching is a social feature on GitHub that allows users to receive notifications on new discussions and events for a given project. Watching is generally used to assess the quality, popularity and worthwhile of a project [29, 19]. In fact, the number of watchers positively correlates with the number of issues and forks [39, 40]. Watching is also used as implicit coordination mechanism [90]. Watchers represent a valuable resource for a project and can influence its health since they represent a pool for recruiting the project's future contributors [91].

5.1.13. Ecosystem - Characterization

Typical ecosystems in GitHub. A software ecosystem is defined as a collection of software projects which are developed and co-evolve together (due to technical dependencies and shared developer communities) in the same environment[92]. Most ecosystems in GitHub revolve around one central project [93], whose purpose is to support software development, such as frameworks and libraries [93, 65, 47] on which the other projects in the ecosystem rely on.

Popular ecosystems are usually interconnected. Most ecosystems are interconnected, while small, unpopular ones remain isolated and tend to contain projects owned by the same GitHub user or organization [93, 89].

JavaScript, Java and Python dominate the ecosystems. In GitHub, there are two clear dominant clusters of programming languages. One cluster is composed by the web programming languages (Java Script, Ruby, PHP, CSS), and a second one revolves around system oriented programming languages (C, C++, Python) [57]. JavaScript,

Java and Python are the top 3 languages in GitHub [55, 56, 57]. In particular, until 2011 Ruby was the dominant programming language on GitHub, and Java Script, Ruby and Python was the top 3 programming languages until 2012. In 2013 and 2014, Java took the second place in 2013 and 2014, after JavaScript [57].

5.1.14. Ecosystem - Transparency

Transparency and users. GitHub's transparency helps in identifying user skills as well as enabling learning from the actions of others, facilitating the coordination between users in a given ecosystem [29, 54, 19, 26, 80], also lowering the barriers for joining specific projects [94, 52, 26, 62].

Transparency and projects. The transparency provided by GitHub can help in making sense of the project (or ecosystem) evolution over time [29]. In particular, the amount of commits, branches, forks as well as open and closed pull requests and issues are meant to reflect the project activity and size and how the project is managed [43, 19]. For instance, lots of open and ignored pull requests signal that external contributions are often not taken into account, since each open pull request indicates an offer of code that is being ignored rather than accepted, rejected or commented upon [29, 19].

5.1.15. Ecosystem - Relationship with other platforms

GitHub and more. Activities around a software development project are not exclusively performed in GitHub [21], but they leverage on other platforms and channels with superior capabilities in terms of social functions, making user connections easier [90]. Thus revealing the existence of an ecosystem on the Internet for software developers which includes many platforms, such as GitHub, Twitter and StackOverflow, among others [90]. In particular, Storey et al. report that developers in GitHub use of average 11.7 communication channels such as face-to-face interactions, Q&A sites, private chats (Skype, Google chat, etc.), mailining lists etc. [55].

GitHub and Twitter. Twitter is used by developers to stay aware of industry changes, for learning, and for building relationships, thus it helps developers keep up with the

fast-paced development landscape [95, 90]. However, Twitter (together with other web sites and search engines) is also used to identify GitHub projects to fork [46].

GitHub and StackOverflow. StackOverflow is often used as a communication channel for project dissemination. In addition, there exists a relation between the users activities across the two platforms [96, 85]. However, while Vasilescu et al. report that active GitHub contributors provide more answers (and ask fewer questions) on StackOverflow, and active StackOverflow answerers make more commits on GitHub [96], Badashian et al. admit that interdependencies between the users activities across the two platforms exists, but the relationship is weak, thus the activity in one platform cannot predict the activity in the other one [85].

5.2. RQ2: Which empirical methods have been used?

In this section we characterize the methodological aspects of the empirical process followed by the selected papers when inferring their results and take a critical look at them to evaluate how confident we can be about those results and how generalizable they might be. We classified the selected papers according to:

- Type of methods employed for the analysis. This dimension can take four values, namely: (1)Metadata observation, when the work relies on the study of Github metadata; (2) surveys; (3) interviews; or (4) a mixture of methods. Other kinds of empirical methods are not added as dimensions since no instance of them were found.

- Sampling techniques used, which classifies the sampling techniques used to build the datasets out of GitHub (i.e., subsets of projects and users input of the analysis phase for the papers). This dimension can take three values, namely: (1) non-probability sampling, (2) probability sampling; or (3) no sampling.

- Self-awareness, which analyzes whether a paper reports on its own limitations (i.e., threats to validity).

Next we report on the classification of the selected papers according to these dimensions.

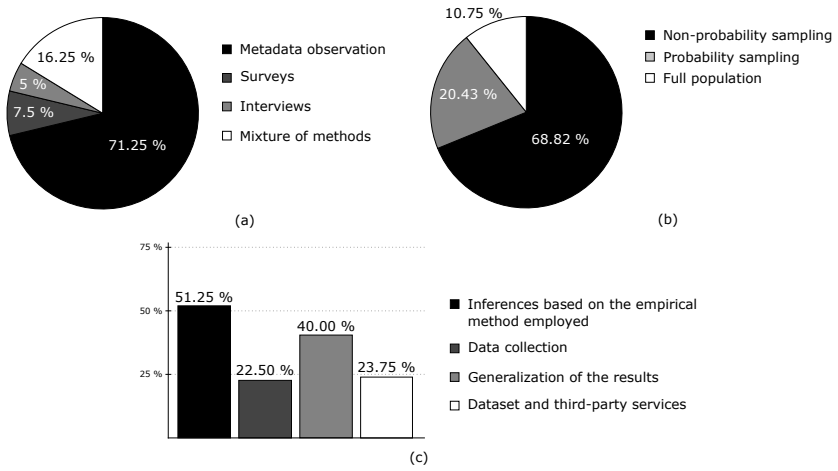


Figure 2: (a) Empirical methods, (b) sampling techniques employed and (c) classification of the limitations reported.

5.2.1. Type of Methods employed

Figure 2a shows the results of the study of empirical methods employed. As can be seen, the great majority of the works (71.25%) rely on the direct observation and mining of GitHub metadata. The use of surveys and interviews was detected in 12.5% of the works, while the remaining 16.25% combine pairs of the previous research methods (e.g., metadata observation and interviews). It is important to note that only 22.5% of the selected works applied longitudinal studies[6], an issue that seems to be common in Open Source studies [97, 21].

5.2.2. Sampling Techniques Used

Figure 2b shows that most of the works (68.82%, 64 works) use non-probability sampling, while around a third (20.43%, 19 works) rely on probability sampling. Interestingly enough, stratified random sampling[7], which takes into account the diversity of projects and users [98], is used just in 11.25% (9) of the works. 10.75% (10) of the

[6]A longitudinal study is a correlational research study that concerns repeated observations of the same variables over long periods of time

[7]Stratified random sampling involves the division of population into smaller groups (strata), that share same characteristics before the sampling takes place.

analyzed works do not use sampling techniques at all.

This range of sampling strategies can bias the generalization of the findings where representativeness is important. For instance, most of the non-probability sampling papers handpick successful projects (in terms of popularity, code contributions, etc.) and therefore cannot be used to represent the average GitHub project. This can be justified in some cases (where, for instance, we want to study specific groups of projects, e.g. only highly popular ones) but not as general rule.

5.2.3. Self-Awareness

By analyzing the limitations self-reported in the selected papers, we can assess their degree of self-awareness regarding potential threats to validity. Interestingly enough, 33.75% of them did not comment on any threat. For the rest, we identified four categories of publicly acknowledged limitations: (1) limitations on the empirical method employed, (2) on the data collection process, (3) on the generalization of the results and (4) on the dataset and use of third-party services. Fig. 2c shows the results. Note that works may report limitations covering several categories.

Around half of the works (51.25% of the works reporting limitations) reported threats due to the empirical method employed, which included potential errors and bias introduced by the authors, techniques and tools used. With respect to the data collection (22.50%), it is interesting to note that around half of them explicitly commented on problems with the GitHub API (e.g., limited quota of requests or events not properly returned). The issues regarding the generalization of the results (40.00%) was mainly due to the non-probability sampling techniques chosen. Instead, generalization of the GitHub results for Open Source in general is not regarded as a threat and assumed to be true given the current dominance of GitHub as code hosting platform. Finally, we detected some works (23.75%) which reported problems with datasets and third-party services mirroring GitHub, mostly related to their size and data freshness.

5.3. RQ3: Which technologies have been used to extract and build datasets from GitHub?

Papers studying GitHub must first collect the data they need to analyze, which can either be created from scratch or by reusing existing datasets. We classified the selected papers according to:

- Data collection process, which reports on the tool/s used to retrieve the data. Currently there are six possible values for this dimension according to the tool used: (1) GHTorrent [79], (2) GitHub Archive, (3) GitHub API, (4) Others (e.g., BOA [99]), (5) manual approach and (6) a mixture of them.
- Dataset size and availability, which reports on the number of users and/or projects of the dataset used in the study, and indicates whether the dataset is provided together with the publication. This may help to evaluate the replicability of the work.

5.3.1. Data collection process

Figure 3a depicts the frequency of each data source, showing that the main ones are GHTorrent and the GitHub API. This figure illustrates a clear trade-off when deciding the data source: curated data vs fresh data. Third party solutions offer a more curated dataset that facilitates the analysis while the GitHub API guarantees an up-to-date information. Moreover, the GitHub API request limit acts as a barrier to get data from GitHub. This affects, both, curated datasets (that take their data raw also via the GitHub API) and individual researchers accessing directly the API.

5.3.2. Datasets size

In total, 36.56% of the works reported the dataset size in terms of projects, while 36.56% of them used the number of users. 26.88% of the works provided the two dimensions. Figures 3 summarizes the number and size of the datasets according to the number of projects (see Figure 3b), number of users (see Figure 3c) and both (see Figure 3d).

Regarding the dataset availability, there are 43.75% (35 papers) of the works providing either a link to download the datasets used or use datasets freely available on

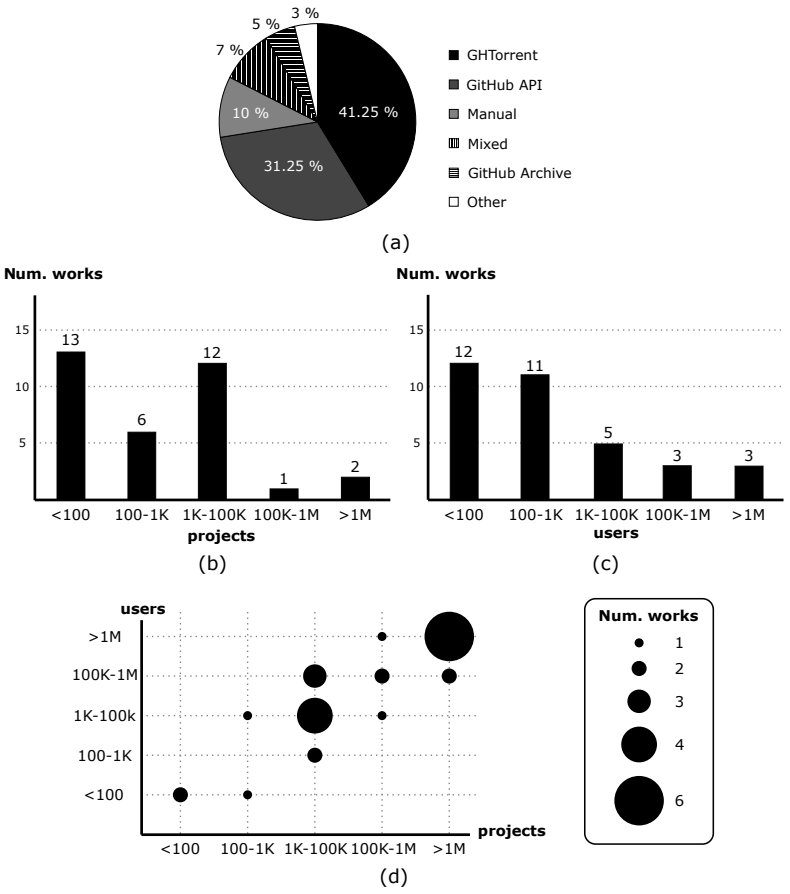


Figure 3: (a) Tools used to collect data from GitHub. Number of works reporting the size of their datasets according to (b) the number of projects, (c) number of users and (d) both.

the Web. The remaining 56.25%, although mostly explaining how the datasets were collected and treated, do not provide any link to the dataset nor to an automatic process to regenerate it.

5.4. RQ4: What are the research communities and the publication fora used?

We believe it is also interesting as part of a systematic mapping study to characterize the research community behind the field, both in terms of the people and the venues where the works are being published. According to this, we analyze the two following dimensions

- Researchers, which we propose to analyze by building the co-authorship graph of the selected papers. In this kind of graphs, authors are represented as nodes while co-authorship is represented as an edge between the involved author nodes. Furthermore, the weight of a node represents the number of papers included in the set of selected papers for the corresponding author while the weight of an edge indicates the number of times the involved author nodes have coauthored a paper. By using co-authorship graphs, we can apply well-known graph metrics to analyze the set of selected papers from a community dimension perspective.

- Publication fora, which requires a straightforward analysis as we only have to use the venue where each selected paper was published.

5.4.1. Researchers

Figure 4 shows the co-authorship graph for the set of authors of the selected papers. The graph includes 179 nodes (i.e., unique authors) and 316 edges (i.e., co-authorship relations). We analyzed the number of connected components in the graph, which helps to identify sets of authors that are mutually reachable through chains of co-authorships. In total, there are 39 connected components and, as can be seen, there are 9 major sub-graphs including more than 5 author nodes. One of them (see bottom part of the graph) is the largest sub-graph including 41 author nodes (almost 22.91% of the total number of authors). We consider each of these connected components as sub-communities of collaboration. This last connected component also contributes to a global graph diameter of 7 (largest distance between two author nodes). On the other hand, the average path length in the graph is 2.94.

We also calculated the betweenness centrality value for each node, which measures the number of shortest paths between any two nodes that pass through a particular node and allows identifying prominent authors in the community that act as bridges between group of authors. The darker a node is in Figure 4, the higher the betweenness centrality it has (i.e., the more shortest path pass through).

Finally, we also measured the graph density, which is the relative fraction of edges in the graph, that is, the ratio between the actual number of edges and the maximum number of possible edges in the graph. When applied to co-authorship graphs, this

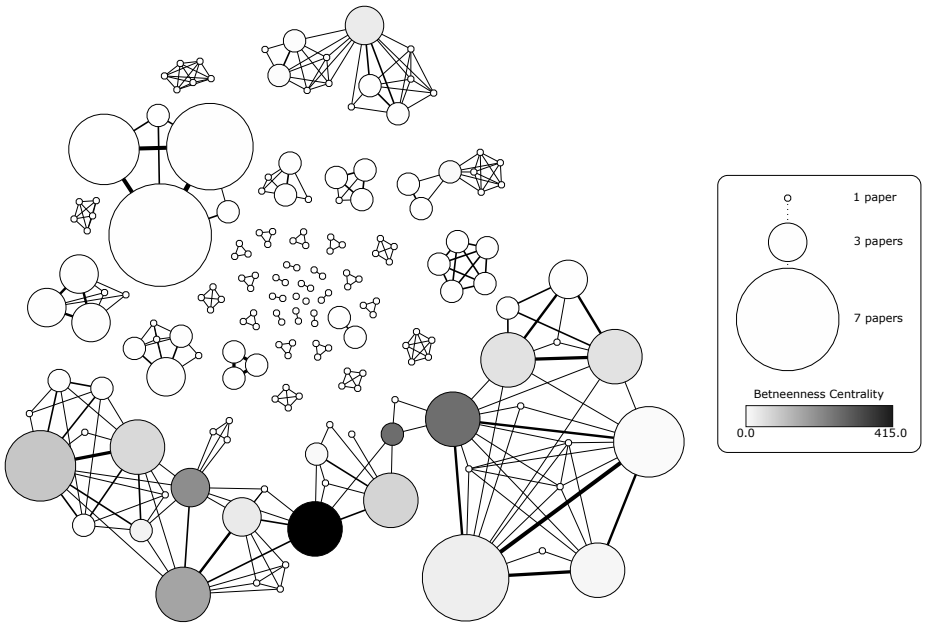


Figure 4: Co-authorship graph for the selected papers. Nodes represent authors and edges represent co-authorships. The bigger the node, the more papers such author has. The thicker the edge, the more papers the involved authors have coauthored. The darker the node, the higher the betweenness centrality.

metric can help us to measure the collaboration by studying how connected are the authors. In our graph, the graph density is 0.02, which is a very low value due to the existence of numerous connected components. This open the door to further collaborations between authors in the different subcomponents to enrich the research results in the area.

5.4.2. Publication fora

Table 6 shows the distribution along the years of the number of papers both collected and finally selected, as well as the publication type for the latter group. They span from 2009 to (July) 2016, and, as can be seen, the number of papers has been definitely increasing. From the selected works, 72.50% are conference papers, 11.25% are technical reports, while journal and workshop papers account for 11.25% and 5.00%, respectively. Figure 5 depicts the number of primary sources for each publication forum.

	Collected	Selected		Techn. rep.	Work.	Conf.	Journ.
2009	0	0	=	0	0	0	0
2010	2	1	=	1	0	0	0
2011	1	0	=	0	0	0	0
2012	15	5	=	1	0	4	0
2013	44	12	=	1	0	10	1
2014	96	25	=	2	2	20	1
2015	108	21	=	0	1	18	2
2016	76	16	=	4	1	6	5
Total	342	80	=	9 (11.25%)	4 (5.00%)	58 (72.50%)	9 (11.25%)

Table 6: Distribution of works along the years.

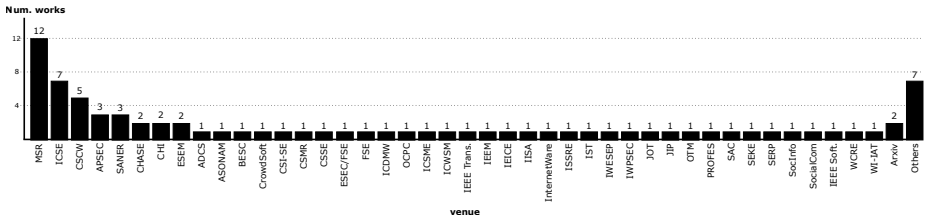


Figure 5: Publications fora summary.

6. Discussion

In this section we highlight and discuss some of the findings from our study. This section extends our early analysis in a short paper [100] of some of these issues (and on a more recent and different selection of papers).

Lack of specific development areas and interdisciplinary studies. Our analysis of areas/topics reveals a lack of works studying GitHub with a more interdisciplinary perspective, e.g. from a political point of view (e.g., study of governance models in OSS), complex systems (e.g., network analysis) or social sciences (e.g., quality of discussions in issue trackers). We also miss works targeting early phases of the development cycle (e.g. studying requirements and design aspects). Obviously, this kind of studies are more challenging since there is typically less data available in GitHub or needs more processing.

Overuse of quantitative analysis. Right now, a vast majority of studies rely only on the analysis of GitHub metadata. We hope to see in the future a better combination of such studies (typically large regarding the spectrum of analyzed projects but shallow in the analysis of each individual project) with other studies targeting the same research question but performing a deeper analysis (e.g. including also interviews to understand the reasons behind those results) for a smaller subset of projects. We detected some evidences of this shift in the number of works applying longitudinal studies, where we found a positive trend, thus improving the situation detected in OSS by other works [97, 21].

Poor sampling techniques. We believe that the GitHub research community could benefit from a set of benchmarks (with predefined sets of GitHub projects chosen and grouped according to different characteristics) and/or from having trustworthy algorithms responsible for generating diverse and representative samples [98] according to a provided set of criteria to study. These samples could, in turn, be stored and made available for further replicability studies.

Small datasets size. Most papers use datasets of small-medium size, which are an important threat to the validity of results when trying to generalize them. Also, more than half of the analyzed paper did not provide a link to the dataset they used nor an automatic process to regenerate it, which may also hamper the replicability of the studies. However, it is important to note that the number of works sharing their datasets is increasing over time.

Low level of self-awareness. We believe the limited acknowledgment of their threats to validity (around 34% do not report any) puts the reader in the very difficult situation of having to decide by herself the confidence in the reported results the work deserves. Works reporting partial results or findings on very concrete datasets are perfectly fine but only as long as this is clearly stated. However, we detected an upward trend in reporting threats to validity in recent years.

Need of replication and comparative studies. Clearly urgent since almost none exist

at the moment. Replicability is also hampered by some of issues above and the difficulties of performing (and publishing) replicability studies in software engineering [101]. Comparative studies need to first set on a fixed terminology. Some papers may seem inconsistent when reporting on a given metric but a closer look may reveal that the discrepancy is due to their different interpretation. For instance, this is common when talking about success, popularity or activity in a project (e.g. one may consider a project successful as equivalent to popular, and by popular mean to be starred a lot, while the other may interpret successful as a project with a high commit frequency).

API restrictions limits the data collection process. GitHub self-imposed limitations on the use of its API hinders the data collection process required to perform wide studies. A workaround can be using OAuth access tokens from different users to collect the information needed. An OAuth access token[8] is a form to interact with the API via automated scripts. It can be generated by a user to allow someone else to make requests on his behalf without revealing his password. Still, researchers may need a large number of tokens for some studies. To deal with this issue, we propose that GitHub either lifts the API request limit for research projects (pre-approving them first if needed) or offers an easy way for individual users to donate their tokens to research works they want to support.

Privacy concerns. Data collection can also bring up privacy issues. Through our study, we discovered that GitHub and third-party services, built on top of it, are sometimes used to contact users (using the email they provide in their GitHub profile) to find potential participants for surveys and interviews [66, 54, 67, 30, 26, 95, 87]. This potential misuse of user email addresses can raise discontent and complaints from the GitHub users [9]. Even when it is not clear whether such complaints are reasonable, the controversy alone may hamper further research works and therefore must be dealt with care. As with the tokens before, we would encourage GitHub to also have an option in the user profile page to let users say whether they allow their public data to be mined

[8] https://developer.github.com/v3/oauth/

[9] For instance: https://github.com/ghtorrent/ghtorrent.org/issues/32

as part of research works and under what conditions.

7. Threats to Validity

The main threats to the validity of this study concern the search process and the selection criteria. In particular, we relied on a set of digital libraries and their query and citation support. Some works might have been ignored due to not being properly indexed in those libraries. To mitigate this issue, we also performed a snowball analysis and an issue-by-issue browsing of top-level conferences and journals in the area to look for additional papers.

We may also have ignored or filter out some relevant works that did not fit in our selection criteria. However, given the large number of retained works after applying the selection process, we believe that the points of discussions we report in Sect. 6 would still be valid.

Focusing the analysis on GitHub could also be regarded as a limitation of this work, though we do not claim the results generalize to other source forges. Still, given the absolute dominance of GitHub as data source for software mining research works (as can be seen by perusing the latest editions of conferences like MSR - Int. Conf. on Mining Software Repositories), we believe our results are representative of this research field.

Another threat to validity we would like to highlight is our subjectivity in screening, classifying and understanding the original authors' point of view of the studied papers. A wrong perception or misunderstanding on our side of a given paper may have resulted in a misclassification of the paper.

To minimize the chances of this to happen, we applied a coding scheme as follows. Data acquisition and analysis & results processes were performed by one coder, who was the first author of this paper. To ensure the quality of both process, a second coder, who was the second author of this paper, validated each phase by randomly selecting a sample of size 25% of the input of the phase (e.g., a sample of 86 papers for the classification phase in the selection process) and performing the phase himself. The intercoder agreement reached for all phases was higher than 93%. We also calculated

the Cohen's Kappa value to calculate the intercode reliability. This index is considered more robust than percent agreement because it takes into account the agreement occurring by chance. The kappa value for all the phases was higher than 0.8, which is normally considered as near complete agreement.

8. Conclusions

In this paper, we have presented a systematic mapping study of the results (and the empirical methods employed to infer those results) reported by all the papers mining software repositories in GitHub to study how software development practices evolve and adapt to the massive use of this social coding platform, specially (but not only) for open source development. Throughout a combination of systematic searches, pruning of non-relevant works and comprehensive forward and backward snowballing processes, we have identified 80 relevant works that have been thoroughly studied to shed some light on the development, project management and community aspects of software development.

We believe this knowledge is useful to project owners, committers and end-users to improve and optimize how they collaborate online and help each other to advance software projects faster and in a way that aligns better with their own interests. These lessons could also be useful for proprietary projects and/or projects happening outside GitHub.

Our analysis also raises some concerns about how reliable are the reported facts given that, overall, papers use small datasets, poor sampling techniques, employ a scarce variety of methodologies and/or are hard to replicate. Although, we also acknowledge a positive shift in the last years. We have offered a few suggestions to mitigate these issues and hope this paper serves also to trigger further discussions on these topics in this (still relatively young) research community.

We would also like to see GitHub itself taking a step forward and getting more involved with the research community, e.g. by offering better access to the platform data for research purposes and even sponsoring research on these topics (similar to what Google, Microsoft and IBM already under different initiatives) since this would

benefit both GitHub and the software community as a whole.

References

[1] M. Squire, Forge++: The changing landscape of floss development, HICSS, 2014, pp. 3266–3275.

[2] S. Chacon, Pro git, Apress, 2009.

[3] P. Bjorn, J. Bardram, G. Avram, L. Bannon, A. Boden, D. Redmiles, C. de Souza, V. Wulf, Global software development in a cscw perspective, CSCW, 2014, pp. 301–304.

[4] M.-A. Storey, C. Treude, A. van Deursen, L.-T. Cheng, The impact of social media on software engineering practices and tools, FoSER, 2010, pp. 359–364.

[5] K. Petersen, R. Feldt, S. Mujtaba, M. Mattsson, Systematic mapping studies in software engineering, EASE, 2008, pp. 68–77.

[6] R. E. Lopez-Herrejon, L. Linsbauer, A. Egyed, A systematic mapping study of search-based software engineering for software product lines, Information and Software Technology 61 (2015) 33 – 51.

[7] A. Zagalsky, J. Feliciano, M.-A. Storey, Y. Zhao, W. Wang, The emergence of GitHub as a collaborative platform for education, CSCW, 2015, pp. 1906–1917.

[8] J. Longo, T. M. Kelley, Use of GitHub as a platform for open collaboration on text documents, OpenSym, 2015, p. 22.

[9] R. C. Davis, Git and GitHub for librarians, Behavioral & Social Sciences Librarian 34 (3) (2015) 158–164.

[10] I. Mergel, Introducing open collaboration in the public sector: The case of social coding on GitHub, Available at SSRN 2497204.

[11] A. Begel, J. Bosch, M.-A. Storey, Social networking meets software development: Perspectives from GitHub, msdn, stack exchange, and topcoder, Software, IEEE 30 (1) (2013) 52–66.

[12] A. Decan, T. Mens, M. Claes, P. Grosjean, On the development and distribution of r packages: An empirical analysis of the r ecosystem, ECSAW, 2015, p. 41.

[13] M. Goeminne, T. Mens, Towards a survival analysis of database framework usage in java projects, ICSME, 2015, pp. 551–555.

[14] R. Venkataramani, A. Gupta, A. Asadullah, B. Muddu, V. Bhat, Discovery of technical expertise from open source code repositories, WWW, 2013, pp. 97–98.

[15] Y. Yu, H. Wang, G. Yin, C. X. Ling, Who should review this pull-request: Reviewer recommendation to expedite crowd collaboration, Vol. 1 of APSEC, 2014, pp. 335–342.

[16] K. Muthukumaran, A. Choudhary, N. Murthy, Mining GitHub for novel change metrics to predict buggy files in software systems, CINE, 2015, pp. 15–20.

[17] L. Zhang, Y. Zou, B. Xie, Z. Zhu, Recommending relevant projects via user behaviour: an exploratory study on GitHub, CrowdSoft, 2014, pp. 25–30.

[18] A. Lima, L. Rossi, M. Musolesi, Coding together at scale: GitHub as a collaborative social network, ICWSM, 2014, p. 10.

[19] L. Dabbish, C. Stuart, J. Tsay, J. Herbsleb, Leveraging transparency, Soft., IEEE 30 (1) (2013) 37–43.

[20] G. Avelino, L. Passos, A. Hora, M. T. Valente, A novel approach for estimating truck factors, ICPC, 2016, p. to appear.

[21] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, D. Damian, An in-depth study of the promises and perils of mining GitHub, Empirical Software Engineering (2015) 1–37.

[22] G. Pinto, I. Steinmacher, M. Aur, et al., More common than you think: An in-depth study of casual contributors, SANER, 2016, pp. 112–123.

[23] K. Yamashita, S. McIntosh, Y. Kamei, A. E. Hassan, N. Ubayashi, Revisiting the applicability of the pareto principle to core development teams in open source software projects, IWPSE, 2015, pp. 46–55.

[24] G. Gousios, M. Pinzger, A. v. Deursen, An exploratory study of the pull-based software development model, ICSE, 2014, pp. 345–355.

[25] G. Gousios, M.-A. Storey, A. Bacchelli, Work practices and challenges in pull-based development: the contributor's perspective, ICSE, 2016, pp. 285–296.

[26] R. Pham, L. Singer, O. Liskin, F. Figueira Filho, K. Schneider, Creating a shared understanding of testing culture on a social coding site, ICSE, 2013, pp. 112–121.

[27] J. Tsay, L. Dabbish, J. Herbsleb, Influence of social and technical factors for evaluating contribution in GitHub, ICSE, 2014, pp. 356–366.

[28] J. Tsay, L. Dabbish, J. Herbsleb, Let's talk about it: evaluating contributions through discussion in GitHub, FSE, 2014, pp. 144–154.

[29] L. Dabbish, C. Stuart, J. Tsay, J. Herbsleb, Social coding in GitHub: transparency and collaboration in an open software repository, CSCW, ACM, 2012, pp. 1277–1286.

[30] G. Gousios, A. Zaidman, M.-A. Storey, A. Van Deursen, Work practices and challenges in pull-based development: the integrator's perspective, Tech. rep., Delft University of Technology, Software Engineering Research Group (2014).

[31] R. Padhye, S. Mani, V. S. Sinha, A study of external community contribution to open-source projects on GitHub, MSR, 2014, pp. 332–335.

[32] J. Marlow, L. Dabbish, J. Herbsleb, Impression formation in online peer production: Activity traces and personal profiles in GitHub, CSCW, 2013, pp. 117–128.

[33] V. J. Hellendoorn, P. T. Devanbu, A. Bacchelli, Will they like this?: Evaluating code contributions with language models, MSR, 2015, pp. 157–167.

[34] D. M. Soares, M. L. de Lima Júnior, L. Murta, A. Plastino, Acceptance factors of pull requests in open-source projects, SAC, 2015, pp. 1541–1546.

[35] C. Casalnuovo, B. Vasilescu, P. Devanbu, V. Filkov, Developer onboarding in GitHub: the role of prior social links and language experience, FSE, 2015, pp. 817–828.

[36] A. Rastogi, N. Nagappan, G. Gousios, Geographical bias in github : perceptions and reality, Tech. rep. (2016).

[37] Y. Yu, H. Wang, V. Filkov, P. Devanbu, B. Vasilescu, Wait for it: Determinants of pull request evaluation latency on GitHub, MSR, 2015, pp. 367–371.

[38] Y. Zhang, G. Yin, Y. Yu, H. Wang, A exploratory study of @-mention in GitHub's pull-requests, APSEC, 2014, pp. 343–350.

[39] T. F. Bissyande, D. Lo, L. Jiang, L. Reveillere, J. Klein, Y. Le Traon, Got issues? who cares about it? a large scale investigation of issue trackers from GitHub, ISSRE, 2013, pp. 188–197.

[40] M. Y. Allaho, W.-C. Lee, Trends and behavior of developers in open collaborative software projects, BESC, 2014, pp. 1–7.

[41] R. Kikas, M. Dumas, D. Pfahl, Issue dynamics in GitHub projects, International Conference on Product-Focused Software Process Improvement, 2015, pp. 295–310.

[42] J. Cabot, J. L. Canovas Izquierdo, V. Cosentino, B. Rolandi, Exploring the use of labels to categorize issues in open-source software projects, SANER, 2015, pp. 550–554.

[43] O. Jarczyk, B. Gruszka, S. Jaroszewicz, L. Bukowski, A. Wierzbicki, GitHub projects. quality analysis of open-source software, SocInfo, 2014, pp. 80–94.

[44] Y. Zhang, H. Wang, G. Yin, T. Wang, Y. Yu, Exploring the use of @-mention to assist software development in GitHub, Internetware, 2015, p. to appear.

[45] J. L. Cánovas Izquierdo, V. Cosentino, J. Cabot, Popularity will not bring more contributions to your oss project, Journal of Object Technology 14 (4).

[46] J. Jiang, D. Lo, J. He, X. Xia, P. S. Kochhar, L. Zhang, Why and how developers fork what from whom in GitHub, ESEM (2016) 1–32.

[47] H. Borges, A. Hora, M. T. Valente, Understanding the factors that impact the popularity of GitHub repositories, ICSME, 2016, p. to appear.

[48] M. M. Rahman, C. K. Roy, An insight into the pull requests of GitHub, MSR, 2014, pp. 364–367.

[49] A. Rastogi, N. Nagappan, Forking and the sustainability of the developer community participation–an empirical investigation on outcomes and reasons, Vol. 1 of SANER, 2016, pp. 102–111.

[50] S. Stanciulescu, S. Schulze, A. Wasowski, Forked and integrated variants in an open-source firmware project, in: ICSME, 2015, pp. 151–160.

[51] D. Celińska, et al., Who is forked on GitHub? collaboration among open source developers, Tech. rep. (2016).

[52] K. Peterson, The GitHub open source development process, Tech. rep. (2013).

[53] F. Chatziasimidis, I. Stamelos, Data collection and analysis of GitHub repositories and users, IISA, 2015, pp. 1–6.

[54] E. Kalliamvakou, D. Damian, K. Blincoe, L. Singer, D. German, Open source-style collaborative development practices in commercial projects using GitHub, ICSE, 2015, pp. 574–585.

[55] M.-A. Storey, A. Zagalsky, L. Singer, D. German, et al., How social and communication channels shape and challenge a participatory culture in software development, IEEE Trans. Softw. Eng.

[56] A. S. Badashian, E. Stroulia, Measuring user influence in GitHub: the million follower fallacy, CSI-SE, 2016, pp. 15–21.

[57] A. Sanatinia, G. Noubir, On GitHub's programming languages, arXiv, 2016.

[58] R. Chen, I. Portugal, Analyzing factors impacting open-source project aliveness, Tech. rep.

[59] Y. Yoshikawa, T. Iwata, H. Sawada, Collaboration on social media: Analyzing successful projects on social coding, arXiv, 2014.

[60] K. Yamashita, Y. Kamei, S. McIntosh, A. E. Hassan, N. Ubayashi, Magnet or sticky? measuring project characteristics from the perspective of developer attraction and retention, JIP 24 (2) (2016) 339–348.

[61] H. Hata, T. Todo, S. Onoue, K. Matsumoto, Characteristics of sustainable oss projects: a theoretical and empirical study, CHASE, 2015, pp. 15–21.

[62] N. McDonald, K. Blincoe, E. Petakovic, S. Goggins, Modeling distributed collaboration on GitHub, ACS 17 (07).

[63] K. Aggarwal, A. Hindle, E. Stroulia, Co-evolution of project documentation and popularity within GitHub, MSR, 2014, pp. 360–363.

[64] S. Weber, J. Luo, What makes an open source code popular on git hub?, ICDMW, 2014, pp. 851–855.

[65] J. Calvo-Villagrán, M. Kukreti, The code following: What builds a following on open source software?, Tech. rep. (2014).

[66] B. Vasilescu, V. Filkov, A. Serebrenik, Perceptions of diversity on GitHub: A user survey (2015) 50–56.

[67] B. Vasilescu, D. Posnett, B. Ray, M. G. van den Brand, A. Serebrenik, P. Devanbu, V. Filkov, Gender and tenure diversity in GitHub teams, CHI, 2015, pp. 3789–3798.

[68] W. Wang, G. Poo-Caamaño, E. Wilde, D. M. German, What is the gist?: understanding the use of public gists on GitHub, MSR, 2015, pp. 314–323.

[69] Z. Wang, D. E. Perry, Role distribution and transformation in open source software project teams, APSEC, 2015, pp. 119–126.

[70] P. Wagstrom, C. Jergensen, A. Sarma, Roles in a networked software development ecosystem: A case study in github, Tech. rep. (2012).

[71] N. Matragkas, J. R. Williams, D. S. Kolovos, R. F. Paige, Analysing the 'biodiversity'of open source ecosystems: the GitHub case, MSR, 2014, pp. 356–359.

[72] S. Onoue, H. Hideaki, A. Monden, K. Matsumoto, Investigating and projecting population structures in open source software projects: A case study of projects in GitHub, IEICE Trans. Inf. Syst. 99 (5) (2016) 1304–1315.

[73] O. M. P. Junior, L. E. Zárate, H. T. Marques-Neto, M. A. Song, B. Horizonte, Using formal concept analysis to study social coding in GitHub, MSR, 2014, pp. 1–6.

[74] J. F. Low, T. Yathog, D. Svetinovic, Software analytics study of open-source system survivability through social contagion, IEEM, 2015, pp. 1213–1217.

[75] E. Guzman, D. Azócar, Y. Li, Sentiment analysis of commit comments in GitHub: an empirical study, MSR, 2014, pp. 352–355.

[76] D. Pletea, B. Vasilescu, A. Serebrenik, Security and emotion: sentiment analysis of security discussions on GitHub, MSR, 2014, pp. 348–351.

[77] Y. Takhteyev, A. Hilts, Investigating the geography of open source software through GitHub, Tech. rep. (2010).

[78] Y. Saito, K. Fujiwara, H. Igaki, N. Yoshida, H. Iida, How do GitHub users feel with pull-based development?, IWESEP, 2016, pp. 7–11.

[79] G. Gousios, D. Spinellis, Ghtorrent: GitHub's data from a firehose, MSR, 2012, pp. 12–21.

[80] J. Marlow, L. Dabbish, Activity traces and signals in software developer recruitment and hiring, CSCW, 2013, pp. 145–156.

[81] J. Jiang, L. Zhang, L. Li, Understanding project dissemination on a social coding site, WCRE, 2013, pp. 132–141.

[82] S. Onoue, H. Hata, K.-i. Matsumoto, A study of the characteristics of developers' activities in GitHub, Vol. 2 of APSEC, 2013, pp. 7–12.

[83] J. T. Tsay, L. Dabbish, J. Herbsleb, Social media and success in open source projects, CSCW, 2012, pp. 223–226.

[84] M. Y. Allaho, W.-C. Lee, Analyzing the social ties and structure of contributors in open source software community, ASONAM, 2013, pp. 56–60.

[85] A. S. Badashian, A. Esteki, A. Gholipour, A. Hindle, E. Stroulia, Involvement, contribution and influence in GitHub and StackOverflow, CSSE, 2014, pp. 19–33.

[86] M. J. Lee, B. Ferwerda, J. Choi, J. Hahn, J. Y. Moon, J. Kim, GitHub developers use rockstars to overcome overflow of news, CHI, 2013, pp. 133–138.

[87] K. Blincoe, J. Sheoran, S. Goggins, E. Petakovic, D. Damian, Understanding the popular users: Following, affiliation influence and leadership on GitHub, IST 70 (2015) 30–39.

[88] J. Xavier, A. Macedo, M. de Almeida Maia, Understanding the popularity of reporters and assignees in the GitHub, SEKE, 2014, pp. 484–489.

[89] Y. Yu, G. Yin, H. Wang, T. Wang, Exploring the patterns of social behavior in GitHub, CrowdSoft, 2014, pp. 31–36.

[90] Y. Wu, J. Kropczynski, P. C. Shih, J. M. Carroll, Exploring the ecosystem of software developers on GitHub and other platforms, CSCW, 2014, pp. 265–268.

[91] J. Sheoran, K. Blincoe, E. Kalliamvakou, D. Damian, J. Ell, Understanding watchers on GitHub, MSR, 2014, pp. 336–339.

[92] M. Lungu, Towards reverse engineering software ecosystems, ICSM, 2008, pp. 428–431.

[93] K. Blincoe, F. Harrison, D. Damian, Ecosystems in GitHub and a method for ecosystem identification using reference coupling, MSR, 2015, pp. 202–207.

[94] F. Thung, T. F. Bissyandé, D. Lo, L. Jiang, Network structure of social coding in GitHub, CSMR, 2013, pp. 323–326.

[95] L. Singer, F. Figueira Filho, M.-A. Storey, Software engineering at the speed of light: how developers stay current using twitter, ICSE, 2014, pp. 211–221.

[96] B. Vasilescu, V. Filkov, A. Serebrenik, Stackoverflow and GitHub: associations between software development and crowdsourced knowledge, SocialCom, 2013, pp. 188–195.

[97] K. Crowston, K. Wei, J. Howison, A. Wiggins, Free/libre open-source software development: What we know and what we do not know, ACM Computing Surveys (CSUR) 44 (2) (2012) 7.

[98] M. Nagappan, T. Zimmermann, C. Bird, Diversity in software engineering research, ESEC/FSE, 2013, pp. 466–476.

[99] R. Dyer, H. A. Nguyen, H. Rajan, T. N. Nguyen, Boa: A language and infrastructure for analyzing ultra-large-scale software repositories, ICSE, 2013, pp. 422–431.

[100] V. Cosentino, J. L. Cánovas Izquierdo, J. Cabot, Findings from GitHub. Methods, Datasets and Limitations, MSR. To appear., 2016.

[101] J. Lung, J. Aranda, S. Easterbrook, G. Wilson, On the difficulty of replicating human subjects studies in software engineering, ICSE, 2008, pp. 191–200.

Figure 1

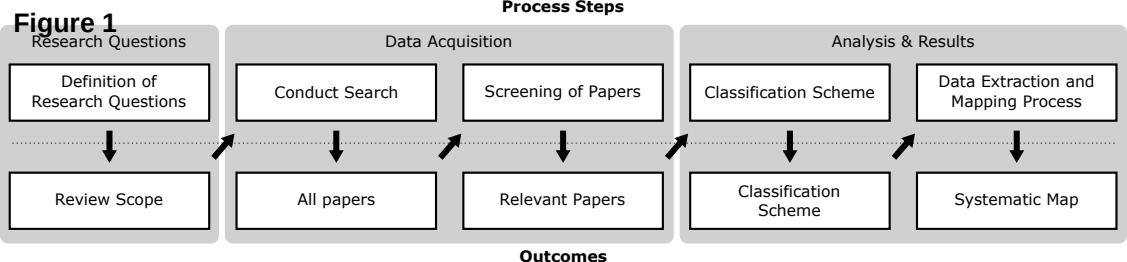
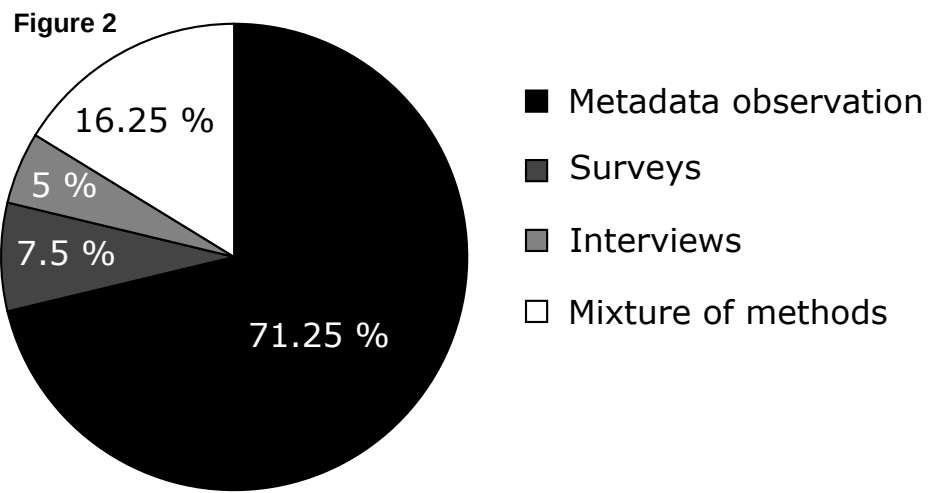
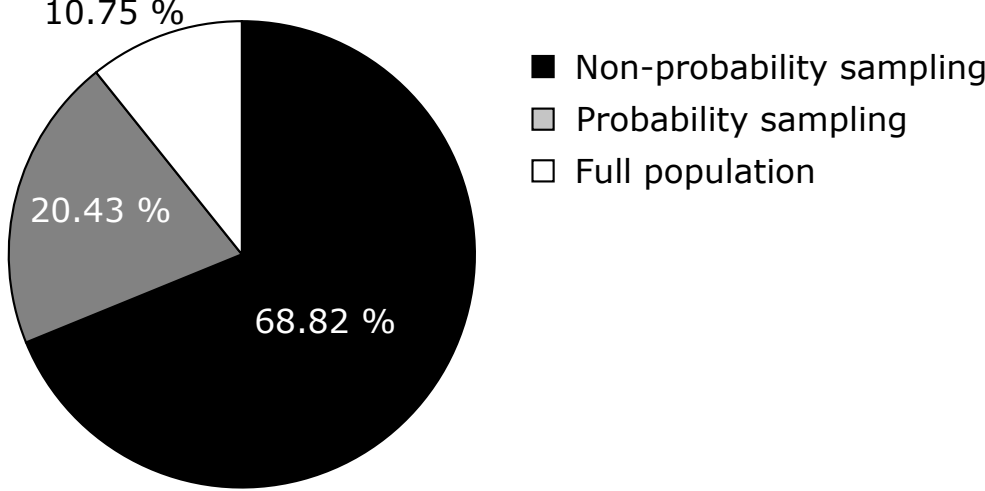

Process Steps
Research Questions
Data Acquisition
Analysis & Results
Definition of Research Questions
Conduct Search
Screening of Papers
Classification Scheme
Data Extraction and Mapping Process
Review Scope
All papers
Relevant Papers
Classification Scheme
Systematic Map
Outcomes

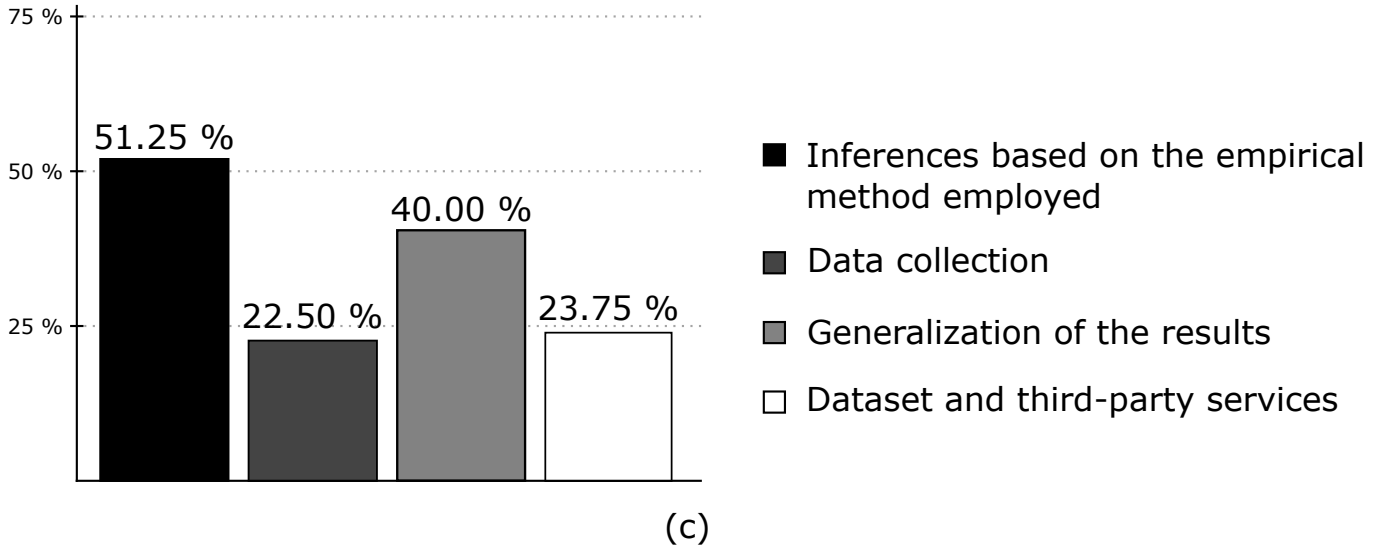
Figure 2


16.25 %
5 %
7.5 %
71.25 %
Metadata observation
Surveys
Interviews
Mixture of methods

(a)

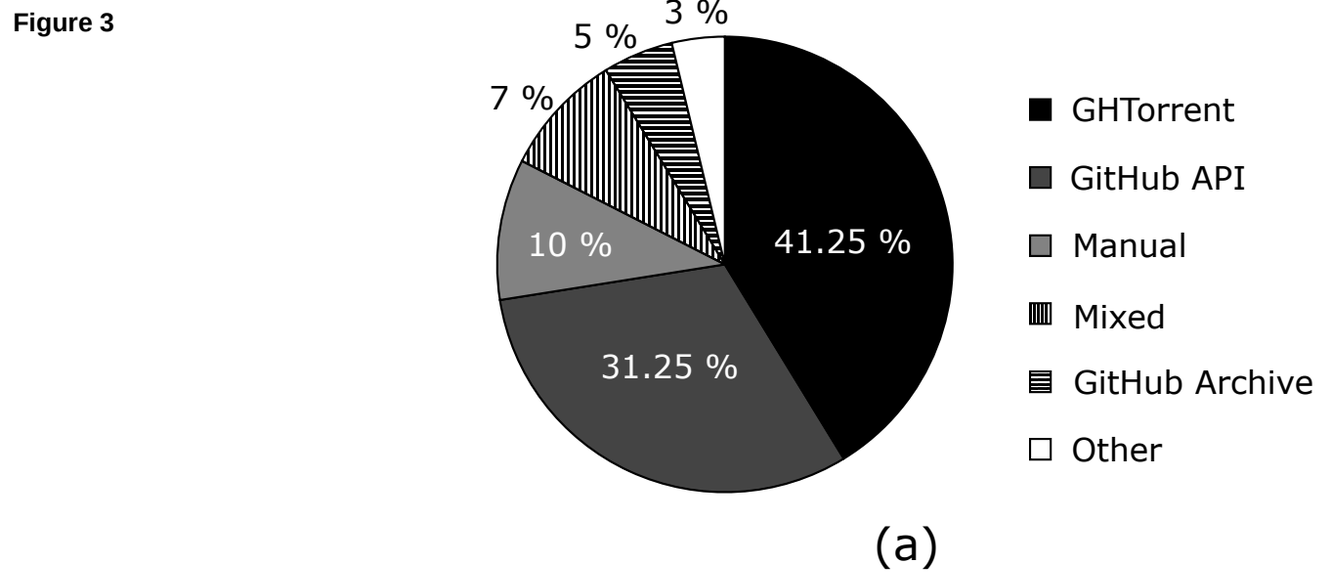

10.75 %
20.43 %
68.82 %
Non-probability sampling
Probability sampling
Full population

(b)

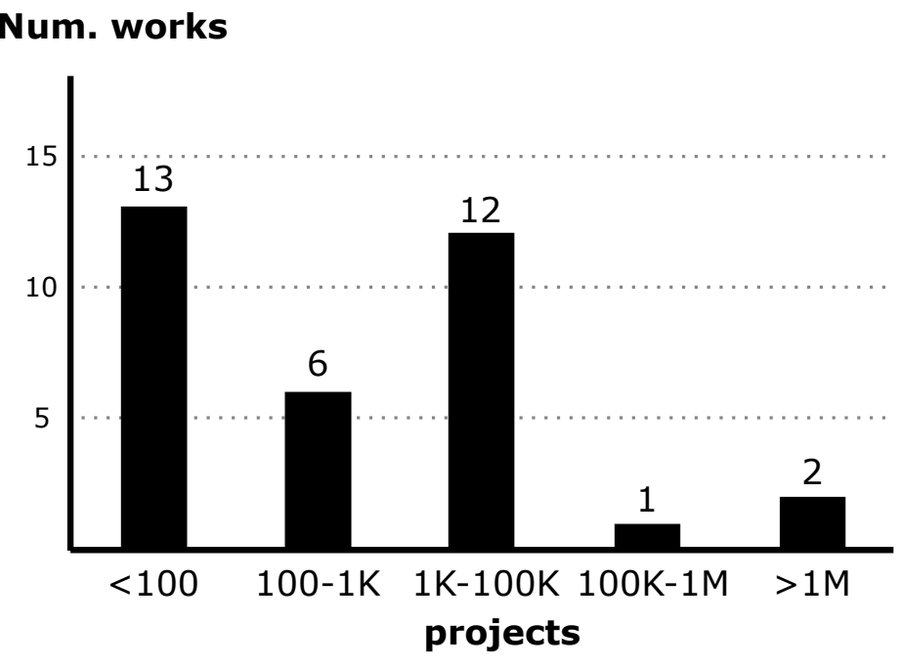

75 %
50 %
25 %
51.25 %
22.50 %
40.00 %
23.75 %
Inferences based on the empirical method employed
Data collection
Generalization of the results
Dataset and third-party services

(c)

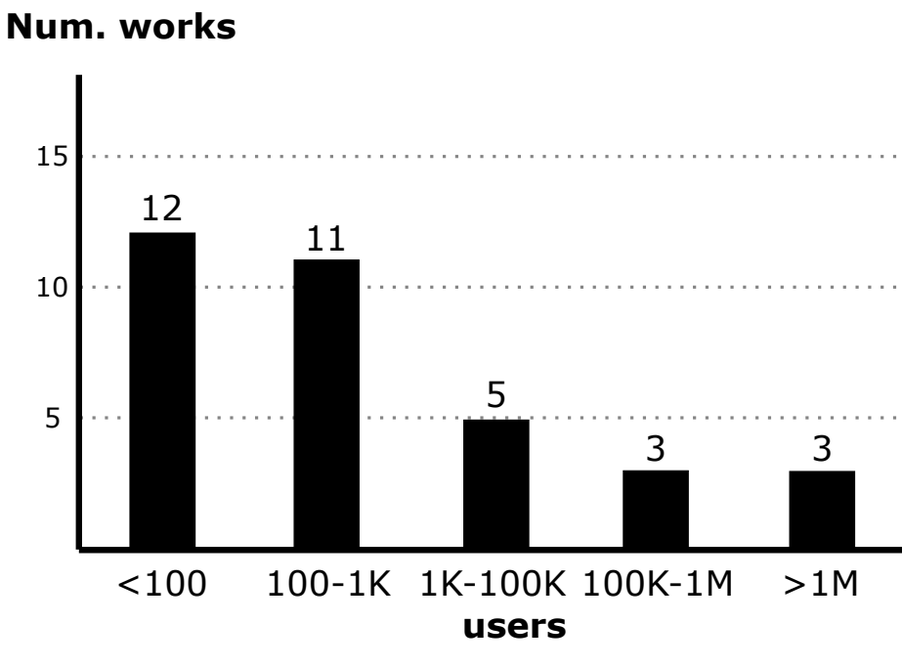
Figure 3



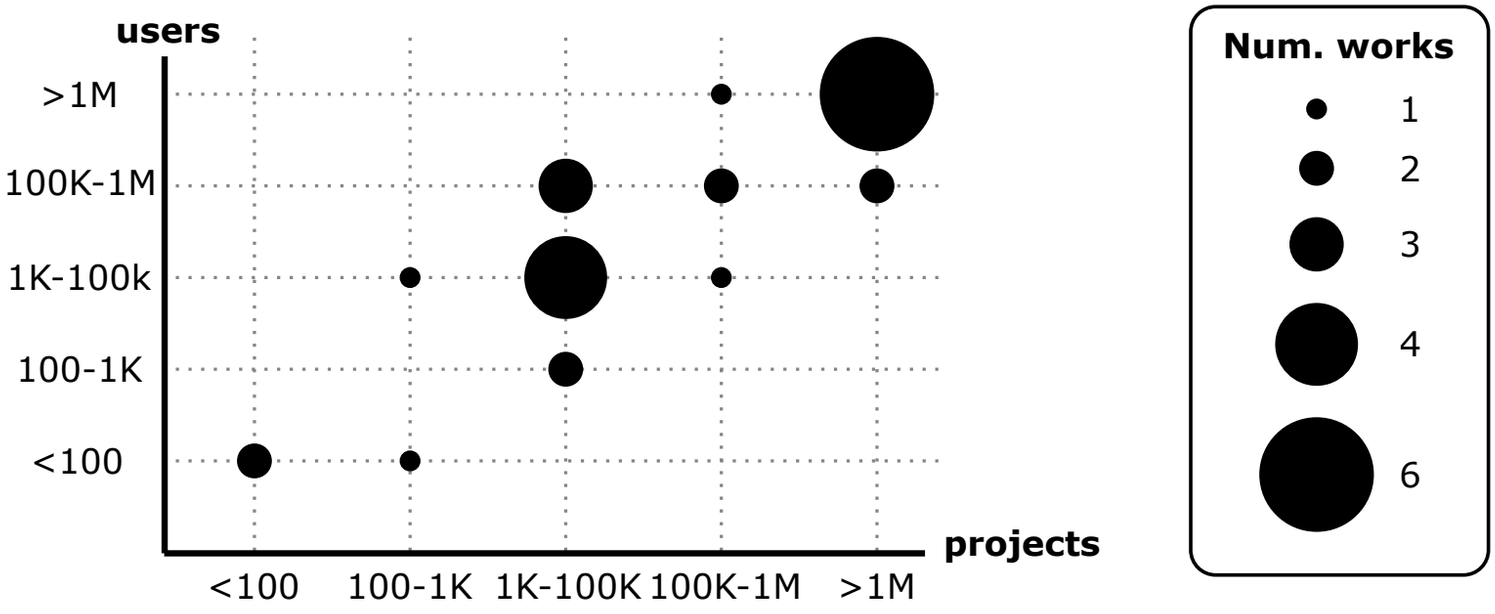
(a)



(b)

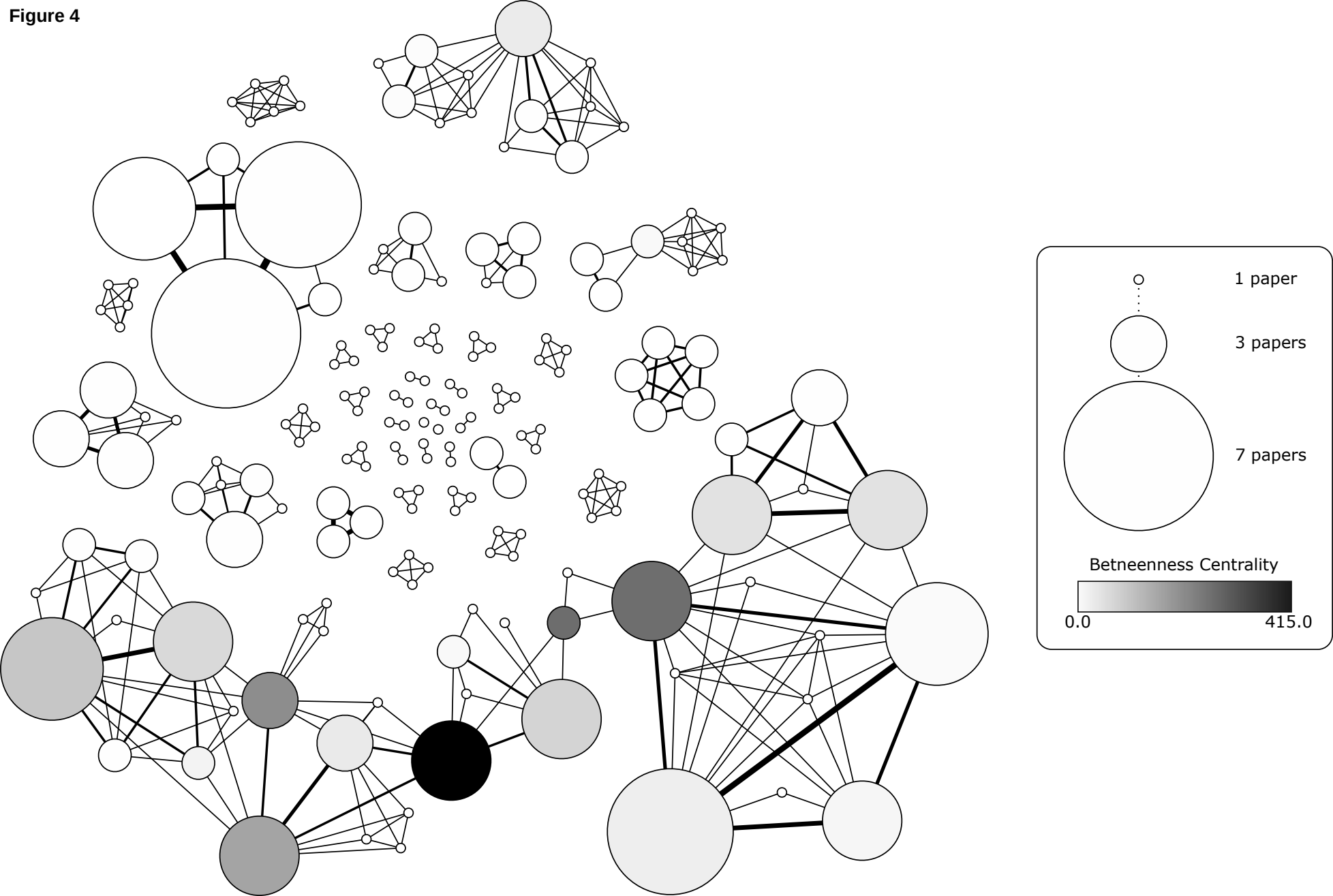


(c)



(d)

Figure 4



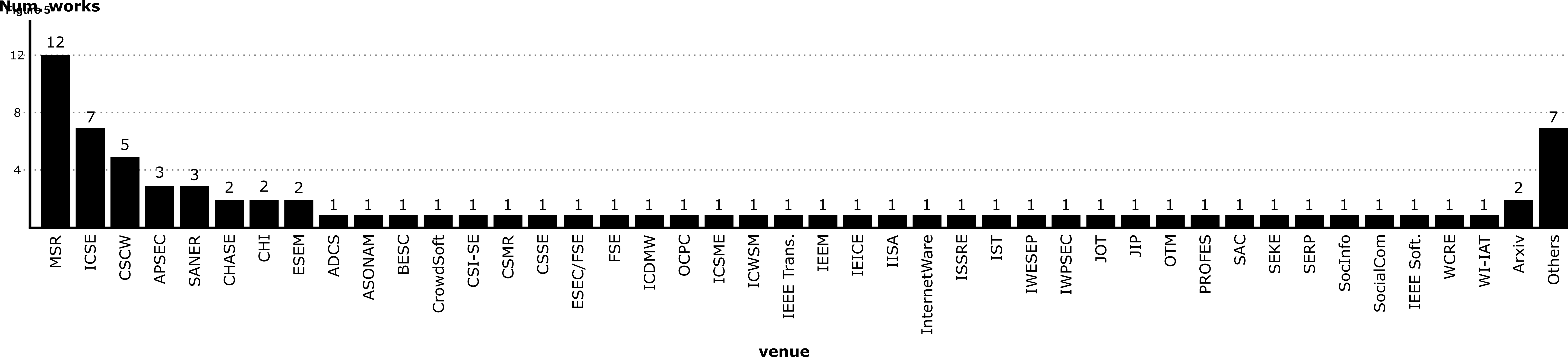


Figure 5
Num. works
12
8
4
12
7
5
3
3
2
2
2
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
2
7
MSR
ICSE
CSCW
APSEC
SANER
CHASE
CHI
ESEM
ADCS
ASONAM
BESC
CrowdSoft
CSI-SE
CSMR
CSSE
ESEC/FSE
FSE
ICDMW
OCPC
ICSME
ICWSM
IEEE Trans.
IEEM
IEICE
IISA
InternetWare
ISSRE
IST
IWESEP
IWPSEC
JOT
JIP
OTM
PROFES
SAC
SEKE
SERP
SocInfo
SocialCom
IEEE Soft.
WCRE
WI-IAT
Arxiv
Others
venue

Table 1

```
This is TeX, Version 3.14159265 (TeX Live 2014/W32TeX) (preloaded
format=tex 2015.9.2)  19 AUG 2016 14:00
**tab-diglib.tex
(./tab-diglib.tex
! Undefined control sequence.
l.3 \begin
          {table}[!t]
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.5 \resizebox
              {\columnwidth}{!}{%
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Missing number, treated as zero.
<to be read again>
                    {
l.5 \resizebox{
               \columnwidth}{!}{%
A number should have been here; I inserted `0'.
(If you can't figure out why I needed to see a number,
look up `weird error' in the index to The TeXbook.)

! Illegal unit of measure (pt inserted).
<to be read again>
                    {
l.5 \resizebox{
               \columnwidth}{!}{%
Dimensions can be in units of em, ex, in, pt, pc,
cm, mm, dd, cc, bp, or sp; but yours is a new one!
I'll assume that you meant to say pt, for printer's points.
To recover gracefully from this error, it's best to
delete the erroneous units; e.g., type `2' to delete
two letters. (See Chapter 27 of The TeXbook.)

! Undefined control sequence.
l.5 \resizebox{\columnwidth
                           }{!}{%
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.6 \begin
          {tabular}{ l l c c c }
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
```

```
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.7 \hline

The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Misplaced alignment tab character &.
l.8 \bf{Digital Library} &
                           \bf{URL} & \bf{Adv.Query} & \bf{Cit. Nav.} &
\bf{...
I can't figure out why you would want to use a tab mark
here. If you just want an ampersand, the remedy is
simple: Just type `I\&' now. But if some right brace
up above has ended a previous alignment prematurely,
you're probably due for more error messages, and you
might try typing `S' now just to see what is salvageable.

! Misplaced alignment tab character &.
l.8 \bf{Digital Library} & \bf{URL} &
                                      \bf{Adv.Query} & \bf{Cit. Nav.} &
\bf{...
I can't figure out why you would want to use a tab mark
here. If you just want an ampersand, the remedy is
simple: Just type `I\&' now. But if some right brace
up above has ended a previous alignment prematurely,
you're probably due for more error messages, and you
might try typing `S' now just to see what is salvageable.

! Misplaced alignment tab character &.
l.8 ...ital Library} & \bf{URL} & \bf{Adv.Query} &
                                                     \bf{Cit. Nav.} &
\bf{Ref....
I can't figure out why you would want to use a tab mark
here. If you just want an ampersand, the remedy is
simple: Just type `I\&' now. But if some right brace
up above has ended a previous alignment prematurely,
you're probably due for more error messages, and you
might try typing `S' now just to see what is salvageable.

! Misplaced alignment tab character &.
l.8 ...bf{URL} & \bf{Adv.Query} & \bf{Cit. Nav.} &
                                                     \bf{Ref. Nav.} \\
I can't figure out why you would want to use a tab mark
here. If you just want an ampersand, the remedy is
simple: Just type `I\&' now. But if some right brace
up above has ended a previous alignment prematurely,
you're probably due for more error messages, and you
might try typing `S' now just to see what is salvageable.

! Argument of \\ has an extra }.
<inserted text>
                \par
```

```
<to be read again>
                   }
l.18 }

I've run across a `}' that doesn't seem to match anything.
For example, `\def\a#1{...}' and `\a}' would produce
this error. If you simply proceed now, the `\par' that
I've just inserted will cause me to report a runaway
argument that might be the root of the problem. But if
your `}' was spurious, just type `2' and it will go away.

Runaway argument?
 \hline Google Scholar & scholar.google.com & - & \checkmark & - \\
D\ETC.
! Paragraph ended before \\ was complete.
<to be read again>
                   \par
<to be read again>
                   }
l.18 }

I suspect you've forgotten a `}', causing me to apply this
control sequence to too much text. How can we recover?
My plan is to forget the whole thing and hope for the best.

! Undefined control sequence.
l.19 \caption
             {Digital libraries selected and main features provided.}
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.20 \label
           {tab:diglib}
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

[1] )
Output written on tab-diglib.dvi (1 page, 436 bytes).
```

Table 2

```
This is TeX, Version 3.14159265 (TeX Live 2014/W32TeX) (preloaded
format=tex 2015.9.2)  19 AUG 2016 14:01
**tab-queries.tex
(./tab-queries.tex
! Undefined control sequence.
l.3 \begin
          {table}[!t]
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.5 \renewcommand
                 {\arraystretch}{1.15}
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Missing number, treated as zero.
<to be read again>
                   {
l.5 \renewcommand{
                  \arraystretch}{1.15}
A number should have been here; I inserted `0'.
(If you can't figure out why I needed to see a number,
look up `weird error' in the index to The TeXbook.)

! Illegal unit of measure (pt inserted).
<to be read again>
                   {
l.5 \renewcommand{
                  \arraystretch}{1.15}
Dimensions can be in units of em, ex, in, pt, pc,
cm, mm, dd, cc, bp, or sp; but yours is a new one!
I'll assume that you meant to say pt, for printer's points.
To recover gracefully from this error, it's best to
delete the erroneous units; e.g., type `2' to delete
two letters. (See Chapter 27 of The TeXbook.)

! Undefined control sequence.
l.5 \renewcommand{\arraystretch
                                 }{1.15}
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.6 \resizebox
              {\columnwidth}{!}{%
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
```

```
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.6 \resizebox{\columnwidth
                           }{!}{%
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.7 \begin
          {tabular}{ll}
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.8 \hline

The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Misplaced alignment tab character &.
l.9 \bf{Digital Library} &
                           \bf{Search query} \\
I can't figure out why you would want to use a tab mark
here. If you just want an ampersand, the remedy is
simple: Just type `I\&' now. But if some right brace
up above has ended a previous alignment prematurely,
you're probably due for more error messages, and you
might try typing `S' now just to see what is salvageable.

! Argument of \\ has an extra }.
<inserted text>
                \par
<to be read again>
                   }
l.34 }

I've run across a `}' that doesn't seem to match anything.
For example, `\def\a#1{...}' and `\a}' would produce
this error. If you simply proceed now, the `\par' that
I've just inserted will cause me to report a runaway
argument that might be the root of the problem. But if
your `}' was spurious, just type `2' and it will go away.

Runaway argument?
 \hline IEEE Xplore & \begin {tabular}{l} \texttt {{{{{"Publication
T\ETC.
! Paragraph ended before \\ was complete.
```

```
<to be read again>
                   \par
<to be read again>
                   }
l.34 }

I suspect you've forgotten a `}', causing me to apply this
control sequence to too much text. How can we recover?
My plan is to forget the whole thing and hope for the best.

! Undefined control sequence.
l.35 \caption
             {Executed queries for each digital library.}
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.36 \label
           {tab:queries}
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

[1] )
Output written on tab-queries.dvi (1 page, 396 bytes).
```

Table 3

```
This is TeX, Version 3.14159265 (TeX Live 2014/W32TeX) (preloaded
format=tex 2015.9.2)  19 AUG 2016 14:01
**tab-venues.tex
(./tab-venues.tex
! Undefined control sequence.
l.3 \begin
          {table}[!t]
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.4 \begin
          {center}
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.5 \resizebox
              {\columnwidth}{!}{%
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.5 \resizebox{\columnwidth
                           }{!}{%
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.6 \begin
          {tabular}{cl}
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.7 \hline

The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.
```

```
! Undefined control sequence.
l.8 \multicolumn
                {2}{c}{\textbf{Conferences}} \\
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.8 \multicolumn{2}{c}{\textbf
                               {Conferences}} \\
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Argument of \\ has an extra }.
<inserted text>
                \par
<to be read again>
                   }
l.40 }

I've run across a `}' that doesn't seem to match anything.
For example, `\def\a#1{...}' and `\a}' would produce
this error. If you simply proceed now, the `\par' that
I've just inserted will cause me to report a runaway
argument that might be the root of the problem. But if
your `}' was spurious, just type `2' and it will go away.

Runaway argument?
 \hline CSCW & Computer-supported cooperative work \\ CSMR &
European\ETC.
! Paragraph ended before \\ was complete.
<to be read again>
                   \par
<to be read again>
                   }
l.40 }

I suspect you've forgotten a `}', causing me to apply this
control sequence to too much text. How can we recover?
My plan is to forget the whole thing and hope for the best.

! Undefined control sequence.
l.41 \caption
             {Selected venues.}
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.42 \label
           {tab:venues}
```

```
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

[1] )
Output written on tab-venues.dvi (1 page, 304 bytes).
```

Table 4

```
This is TeX, Version 3.14159265 (TeX Live 2014/W32TeX) (preloaded
format=tex 2015.9.2)  19 AUG 2016 14:01
**tab-collectionselection.tex
(./tab-collectionselection.tex
! Undefined control sequence.
l.3 \begin
          {table}[!t]
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.5 \resizebox
              {0.75\columnwidth}{!}{%
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Missing number, treated as zero.
<to be read again>
                   {
l.5 \resizebox{
               0.75\columnwidth}{!}{%
A number should have been here; I inserted `0'.
(If you can't figure out why I needed to see a number,
look up `weird error' in the index to The TeXbook.)

! Illegal unit of measure (pt inserted).
<to be read again>
                   {
l.5 \resizebox{
               0.75\columnwidth}{!}{%
Dimensions can be in units of em, ex, in, pt, pc,
cm, mm, dd, cc, bp, or sp; but yours is a new one!
I'll assume that you meant to say pt, for printer's points.
To recover gracefully from this error, it's best to
delete the erroneous units; e.g., type `2' to delete
two letters. (See Chapter 27 of The TeXbook.)

! Undefined control sequence.
l.5 \resizebox{0.75\columnwidth
                                }{!}{%
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.6 \begin
          {tabular}{c|l|r|r|r}
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
```

```
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.7 \hline

The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.8 \multicolumn
                {5}{c}{\textbf{Search for primary sources}}
\...
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.8 \multicolumn{5}{c}{\textbf
                               {Search for primary sources}}
\...
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Argument of \\ has an extra }.
<inserted text>
                \par
<to be read again>
                   }
l.36 }

I've run across a `}' that doesn't seem to match anything.
For example, `\def\a#1{...}' and `\a}' would produce
this error. If you simply proceed now, the `\par' that
I've just inserted will cause me to report a runaway
argument that might be the root of the problem. But if
your `}' was spurious, just type `2' and it will go away.

Runaway argument?
 \hline Phase & Description & Initial Set & Pruning & Final Set \\ \hline
\ETC.
! Paragraph ended before \\ was complete.
<to be read again>
                   \par
<to be read again>
                   }
l.36 }

I suspect you've forgotten a `}', causing me to apply this
control sequence to too much text. How can we recover?
```

```
My plan is to forget the whole thing and hope for the best.

! Undefined control sequence.
l.37 \caption
             {Number of works considered in the search and
selection/screeni...
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.38 \label
           {tab:collectionSelection}
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

[1] )
Output written on tab-collectionselection.dvi (1 page, 400 bytes).
```

Table 5

```
This is TeX, Version 3.14159265 (TeX Live 2014/W32TeX) (preloaded
format=tex 2015.9.2)  19 AUG 2016 14:01
**tab-works.tex
(./tab-works.tex
! Undefined control sequence.
l.3 \begin
          {table}[!t]
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.4 \begin
          {center}
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.5 \begin
          {tabular}{ c c c c }
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.6 \hline

The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Misplaced alignment tab character &.
l.7 {\bf Areas} &
                  {\bf Topics} & {\bf In topic} & {\bf In area}  \\
\hline
I can't figure out why you would want to use a tab mark
here. If you just want an ampersand, the remedy is
simple: Just type `I\&' now. But if some right brace
up above has ended a previous alignment prematurely,
you're probably due for more error messages, and you
might try typing `S' now just to see what is salvageable.

! Misplaced alignment tab character &.
l.7 {\bf Areas} & {\bf Topics} &
                                  {\bf In topic} & {\bf In area}  \\
\hline
I can't figure out why you would want to use a tab mark
here. If you just want an ampersand, the remedy is
simple: Just type `I\&' now. But if some right brace
```

```
up above has ended a previous alignment prematurely,
you're probably due for more error messages, and you
might try typing `S' now just to see what is salvageable.

! Misplaced alignment tab character &.
l.7 {\bf Areas} & {\bf Topics} & {\bf In topic} &
                                                    {\bf In area}  \\
\hline
I can't figure out why you would want to use a tab mark
here. If you just want an ampersand, the remedy is
simple: Just type `I\&' now. But if some right brace
up above has ended a previous alignment prematurely,
you're probably due for more error messages, and you
might try typing `S' now just to see what is salvageable.

)
Runaway argument?
 \hline \multirow {3}{*}{Software development} & Code contributions
&\ETC.
! File ended while scanning use of \\.
<inserted text>
                \par
<*> tab-works.tex

I suspect you have forgotten a `}', causing me
to read past where you wanted me to stop.
I'll try to recover; but if the error is serious,
you'd better type `E' or `X' now and fix your file.

! Emergency stop.
<*> tab-works.tex

*** (job aborted, no legal \end found)

No pages of output.
```

Table 6

```
This is TeX, Version 3.14159265 (TeX Live 2014/W32TeX) (preloaded
format=tex 2015.9.2)  19 AUG 2016 14:01
**tab-pubtype.tex
(./tab-pubtype.tex
! Undefined control sequence.
l.3 \begin
          {table}[!t]
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.5 \resizebox
              {\textwidth}{!}{%
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Missing number, treated as zero.
<to be read again>
                   {
l.5 \resizebox{
               \textwidth}{!}{%
A number should have been here; I inserted `0'.
(If you can't figure out why I needed to see a number,
look up `weird error' in the index to The TeXbook.)

! Illegal unit of measure (pt inserted).
<to be read again>
                   {
l.5 \resizebox{
               \textwidth}{!}{%
Dimensions can be in units of em, ex, in, pt, pc,
cm, mm, dd, cc, bp, or sp; but yours is a new one!
I'll assume that you meant to say pt, for printer's points.
To recover gracefully from this error, it's best to
delete the erroneous units; e.g., type `2' to delete
two letters. (See Chapter 27 of The TeXbook.)

! Undefined control sequence.
l.5 \resizebox{\textwidth
                         }{!}{%
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.6 \begin
          {tabular}{l c  c  c  c  c  c}
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
```

```
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.7 \hline

The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Misplaced alignment tab character &.
l.8      &

I can't figure out why you would want to use a tab mark
here. If you just want an ampersand, the remedy is
simple: Just type `I\&' now. But if some right brace
up above has ended a previous alignment prematurely,
you're probably due for more error messages, and you
might try typing `S' now just to see what is salvageable.

! Misplaced alignment tab character &.
l.9      {\bf Collected} &

I can't figure out why you would want to use a tab mark
here. If you just want an ampersand, the remedy is
simple: Just type `I\&' now. But if some right brace
up above has ended a previous alignment prematurely,
you're probably due for more error messages, and you
might try typing `S' now just to see what is salvageable.

! Misplaced alignment tab character &.
l.10      {\bf Selected} &

I can't figure out why you would want to use a tab mark
here. If you just want an ampersand, the remedy is
simple: Just type `I\&' now. But if some right brace
up above has ended a previous alignment prematurely,
you're probably due for more error messages, and you
might try typing `S' now just to see what is salvageable.

! Misplaced alignment tab character &.
l.11      &

I can't figure out why you would want to use a tab mark
here. If you just want an ampersand, the remedy is
simple: Just type `I\&' now. But if some right brace
up above has ended a previous alignment prematurely,
you're probably due for more error messages, and you
might try typing `S' now just to see what is salvageable.

! Misplaced alignment tab character &.
l.12      {\bf Techn. rep.}  &

I can't figure out why you would want to use a tab mark
here. If you just want an ampersand, the remedy is
simple: Just type `I\&' now. But if some right brace
```

```
up above has ended a previous alignment prematurely,
you're probably due for more error messages, and you
might try typing `S' now just to see what is salvageable.

! Misplaced alignment tab character &.
l.13      {\bf Work.}        &

I can't figure out why you would want to use a tab mark
here. If you just want an ampersand, the remedy is
simple: Just type `I\&' now. But if some right brace
up above has ended a previous alignment prematurely,
you're probably due for more error messages, and you
might try typing `S' now just to see what is salvageable.

! Misplaced alignment tab character &.
l.14      {\bf Conf.}        &

I can't figure out why you would want to use a tab mark
here. If you just want an ampersand, the remedy is
simple: Just type `I\&' now. But if some right brace
up above has ended a previous alignment prematurely,
you're probably due for more error messages, and you
might try typing `S' now just to see what is salvageable.

! Argument of \\ has an extra }.
<inserted text>
                \par
<to be read again>
                   }
l.29 }

I've run across a `}' that doesn't seem to match anything.
For example, `\def\a#1{...}' and `\a}' would produce
this error. If you simply proceed now, the `\par' that
I've just inserted will cause me to report a runaway
argument that might be the root of the problem. But if
your `}' was spurious, just type `2' and it will go away.

Runaway argument?
 \hline 2009 & 0 & 0 & = & 0 & 0 & 0 & 0 \\ 2010 & 2 & 1 & = & 1 & 0
\ETC.
! Paragraph ended before \\ was complete.
<to be read again>
                   \par
<to be read again>
                   }
l.29 }

I suspect you've forgotten a `}', causing me to apply this
control sequence to too much text. How can we recover?
My plan is to forget the whole thing and hope for the best.

! Undefined control sequence.
l.30 \caption
             {Distribution of works along the years.}
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
```

```
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

! Undefined control sequence.
l.31 \label
           {tab:pubtype}
The control sequence at the end of the top line
of your error message was never \def'ed. If you have
misspelled it (e.g., `\hobx'), type `I' and the correct
spelling (e.g., `I\hbox'). Otherwise just continue,
and I'll forget about whatever was undefined.

[1] )
Output written on tab-pubtype.dvi (1 page, 432 bytes).
```

main

Click here to download LaTeX Source Files: github.tex

01-introduction

Click here to download LaTeX Source Files: github-01-introduction.tex

02-methodology

Click here to download LaTeX Source Files: github-02-methodology.tex

03-researchQuestions

Click here to download LaTeX Source Files: github-03-researchQuestions.tex

04-dataAcquisition

Click here to download LaTeX Source Files: github-04-dataAcquisition.tex

05-analysisResults

Click here to download LaTeX Source Files: github-05-analysisResults.tex

06-discussion

Click here to download LaTeX Source Files: github-06-discussion.tex

07-threats

Click here to download LaTeX Source Files: github-07-threats.tex

08-conclusions

Click here to download LaTeX Source Files: github-08-conclusions.tex