Noname manuscript No.
(will be inserted by the editor)

The Promises and Perils of Mining GitHub (Extended Version)

Eirini Kalliamvakou · Georgios Gousios · Kelly Blincoe · Leif Singer · Daniel M. German · Daniela Damian

the date of receipt and acceptance should be inserted later

Abstract With over 10 million `git` repositories, GitHub is becoming one of the most important sources of software artifacts on the Internet. Researchers mine the information stored in GitHub's event logs to understand how its users employ the site to collaborate on software, but so far there have been no studies describing the quality and properties of the available GitHub data. We document the results of an empirical study aimed at understanding the characteristics of the repositories and users in GitHub; we see how users take advantage of GitHub's main features and how their activity is tracked on GitHub and related datasets to point out misalignment between the real and mined data. Our results indicate that while GitHub is a rich source of data on software development, mining GitHub for research purposes should take various potential perils into consideration. For example, we show that the ma-

This work is an extended version of Kalliamvakou et al (2014b), published at the International Working Conference on Mining Software Repositories, MSR 2014. The extension falls into two categories. First, we have identified four additional perils: two are related to the information about GitHub users and two are related to the information that GitHub makes available. Second, we have assessed the perils' potential impact as threats to validity on selected studies.

Eirini Kalliamvakou
University of Victoria E-mail: ikaliam@uvic.ca

Georgios Gousios
Radboud University of Nijmegen E-mail: g.gousios@cs.ru.nl

Kelly Blincoe
University of Victoria E-mail: kblincoe@acm.org

Leif Singer
University of Victoria E-mail: lsinger@uvic.ca

Daniel German
University of Victoria E-mail: dmg@uvic.ca

Daniela Damian
University of Victoria E-mail: danielad@uvic.ca

jority of the projects are personal and inactive, and that almost 40% of all pull requests do not appear as merged even though they were. Also, approximately half of GitHub's registered users do not have public activity, while the activity of GitHub users in repositories is not always easy to pinpoint. We use our identified perils to see if they can pose validity threats; we review selected papers from the MSR 2014 Mining Challenge and see if there are potential impacts to consider. We provide a set of recommendations for software engineering researchers on how to approach the data in GitHub.

Keywords Mining software repositories · `git` · GitHub · code reviews

1 Introduction

GitHub is a collaborative code hosting site built on top of the `git` version control system. It includes a variety of features that encourage teamwork and continued discussion over the life of a project. GitHub uses a "fork & pull" model where developers create their own copies of a repository and submit requests when they want the project maintainer to pull their changes into the main branch, thus providing an environment in which people can easily conduct code reviews. Every repository can optionally use GitHub's issue tracking system to report and discuss bugs and other concerns. GitHub also contains integrated social features: users are able to subscribe to information by "watching" projects and "following" other users, resulting in a constant stream of updates about people and projects of interest. The system supports user profiles that provide a summary of a person's recent activity within the site, such as their commits, the projects they forked or the issues they reported.

Promise I: GitHub is a rich source of software engineering research

With over 10.6 million repositories[1] hosted as of January 2014, GitHub is currently the largest code hosting site in the world. Software engineering researchers have been drawn to GitHub due to this popularity, as well as its integrated social features and the metadata that can be accessed through its API. To date, there has been a variety of research on GitHub and its community. Qualitative studies (Begel et al, 2013; Dabbish et al, 2012; Marlow et al, 2013; McDonald and Goggins, 2013; Pham et al, 2013; Gousios et al, 2015) have focused on how developers use GitHub's social features to form impressions of and draw conclusions about developer and project activity to assess success, performance, and possible collaboration opportunities. Quantitative studies (Gousios et al, 2014; Takhteyev and Hilts, 2010; Thung et al, 2013; Tsay et al, 2012) have attempted to systematically archive GitHub's publicly available data and use it to investigate development practices and network structure in the GitHub environment.

[1] `https://github.com/features`

As part of our research about collaboration on GitHub (Kalliamvakou et al, 2014a), we conducted an exploratory online survey in 2013 to assess the reasons for developers using GitHub and how it supports them in working with others. While analyzing the survey data, we noticed that GitHub repositories were also used for purposes other than strictly software development: many respondents were using repositories to archive data, to host personal projects without any plans to collaborate on their work, or for activities outside of software engineering. This signaled that there may be significant unseen perils in using GitHub data "as-is" for software engineering research. The variety of repository contents and activity, as well as developers' intentions, can alter research conclusions if care is not taken to first establish that the data fits the research purpose.

The potential for misinterpretation when working with publicly mined data has also been noted with datasets pulled from SourceForge (Howison and Crowston, 2004). Furthermore, Bird et al (2009b) described the promises associated with exploiting the information stored in a decentralized version control system. Due to these issues, we formulated the following research question:

RQ: What are the promises and perils of mining GitHub for software engineering research?

This study highlights potential threats to validity for research that relies on GitHub as the main source of data about software engineering development. We use insights gained from a survey conducted with 240 GitHub users to identify potential perils, and we provide evidence of these perils based on quantitative analysis of the GHTorrent dataset as well as a manual inspection of 434 GitHub repositories. We outline some analysis risks to avoid and provide recommendations on how researchers can best use the data available from GitHub. To demonstrate the usefulness of these perils, we analyzed four papers from the MSR'14 Mining Challenge(Baysal and Gousios, 2014). We described how the perils appear in the dataset used in this challenge and create potential validity threats to the results of these papers.

2 Background & Related Work

Many of the projects hosted on GitHub are public, allowing anyone with an Internet connection to view the activity within those projects, including information about issues, pull requests, commits, comments and subscriptions. The large amount of public data on GitHub and its availability via an API make it possible for researchers to easily mine project data. Various tools and datasets have been created to assist researchers with this task.

2.1 Background

Web-based code hosting services such as GitHub have piqued the interest of many a software engineering researcher. The abundance and availability of public data simplifies the data collection and processing issues that are often encountered in research. However, there are still practical difficulties that can potentially alter conclusions drawn from the data.

SourceForge is one of the most popular code hosting sites, but it peaked in popularity prior to GitHub's wide-spread adoption (Finley, 2011). Howison and Crowston (2004) noted that projects hosted on SourceForge were often abandoned and that their data was often contaminated with data imported from previous systems. They also found that information was often missing due to project data being hosted outside of the SourceForge space. Similarly, Weiss (2005) concluded that not all SourceForge data is to be considered perfect: names of categories often change in SourceForge and projects are constantly initiated and then go inactive. By comparing his data to that of FLOSSMole[2], Weiss highlighted that information about inactive and inaccessible projects was missing altogether. Rainer and Gale (2005) conducted an in-depth analysis of the quality of SourceForge data. They noted that only 1% of SourceForge projects were actually active as indicated by their metrics. The authors suggested caution when using SourceForge data and advised that the research community should perform an evaluation of the quality of data taken from portals such as SourceForge. Accordingly, we present study findings highlighting potential risks for researchers to keep in mind when drawing conclusions from GitHub data.

Recent software engineering research has also highlighted biases in bug-fix datasets. These biases can compromise the validity and generalizability of studies using the datasets. Researchers often rely on links between bugs and commits made in commit logs, but linked bugs represent only a fraction of the entire population of fixed bugs. Bird et al (2009a) found that this set of bugs is a biased sample of the entire population. Bachmann et al (2010) found that the set of bugs in a bug tracking system itself may be biased since not all bugs are reported through those systems. Nguyen et al (2010) discovered that similar biases exist even in commercial projects that employ strict guidelines and processes. However, (Rahman et al, 2013) showed that a large sample size can counter the effects of bias. In our work, we show that bias exists across large GitHub datasets and provide recommendations on how to avoid such biases.

Bird et al (2009b) described the problems that mining `git` poses for software engineering research. Their work demonstrated that the differences between centralized version control systems (such as `subversion`) and `git` created certain challenges for those using `git` repositories for research.

2.2 Related Work

The introduction of social features in code hosting sites has drawn much attention from researchers. Several qualitative studies have interviewed GitHub

[2] A collection of open source software data, formerly known as OssMole.

users to better understand how these social features are being used (Begel et al, 2013; Dabbish et al, 2012; Marlow et al, 2013). Their findings indicate that

GitHub users form impressions of and draw conclusions about the activities and potential of developers and projects. Users then internalize those conclusions to decide whom and what to keep track of, or where to contribute next. The transparency brought about by these social features also appears to allow teams to maintain awareness of their members' activity and use this towards organizing their work. Pham et al (2013) investigated whether the higher visibility of developer actions enabled by GitHub's social features has an influence on developers' testing behaviors. Through interviews and an online survey, they highlighted the challenges of promoting a desirable testing culture among contributors and suggested strategies for doing so.

Tsay et al (2012) performed a quantitative study on 5,000 projects to understand how GitHub's social features impact project success. McDonald and Goggins (2013) interviewed GitHub users to identify how they measured success on their projects. Their study shows that project members see GitHub's social features as the driver behind increased contribution.

Additional research has extended beyond GitHub's social features. Thung et al (2013) built social networks of developers involved with 100,000 GitHub projects to demonstrate the social structure of the GitHub ecosystem. Takhteyev and Hilts (2010) looked at the geographic locations of GitHub developers by examining self-reported location information available within GitHub profiles. Gousios et al (2014) examined how pull requests work on GitHub. They found that the pull request model offers fast turnaround, increased opportunities for community engagement, and decreased time to incorporate contributions. They showed that a relatively small number of factors affect both the decision to merge a pull request and the time to process it. They also qualitatively examined the reasons for pull request rejection and found that technical reasons are a small minority. They also demonstrated that many pull requests that appear to be unmerged in GitHub were actually merged. This paper extends their work.

Other research has focused on making the data available through the GitHubAPI more readily accessible. The GHTorrent (Gousios and Spinellis, 2012) project provides a mirror of the GitHubAPI data, which it obtains by monitoring and recording GitHub events as they occur and applying recursive dependency-based retrieval of the related resources. When run in standalone mode, GHTorrent can also retrieve the history of individual repositories. Gousios and Zaidman (2014a) have combined this dataset with their research in Gousios et al (2014) to provide a dataset of pull requests for GitHub projects.

The GitHub archive (Grigorik, 2012) provides a dataset of the history of events in GitHub. It also obtains its data by monitoring the GitHub timeline. However, as the GitHub archive started data collection in 2011, it is an incomplete mirror—GHTorrent, in comparison, has retrieved the complete history of GitHub. Moreover, one can use tools such as Gitminer (Wagstrom et al, 2013) to extract the history of events for specific repositories. Gitminer crawls the GitHubAPI for any desired project and produces a graph dataset.

Tsay et al (2014) mined the data they needed for their research using GitHub's API. They described that in order to reach meaningful conclusions, they had to filter out the majority of projects in GitHub because many were inactive, had very few contributors or did not use GitHub's issue tracking system.

Researchers have taken advantage of the large amount of data available from GitHub and tools like the GHTorrent, the GitHub archive and Gitminer to perform studies across a large number of projects. Studies have investigated testing patterns(Kochhar et al, 2013), programming languages mycite-bissyande2013popularity, issue reporting(Bissyande et al, 2013), project success(Tsay et al, 2012), and more. Takhteyev and Hilts (2010) looked at the geographic locations of GitHub developers by examining self-reported location information available within GitHub profiles.

3 Study Design

The detailed analysis reported in this paper was motivated by our own study of the GitHub environment with the goal of examining how it is used for collaboration (Kalliamvakou et al, 2014a)—we surveyed GitHub users and then conducted interviews to further explore our study findings. We selected survey participants from GitHub's public event stream in May 2013, choosing recently active users with public email addresses. Our survey was exploratory with open-ended questions asking about reasons for using GitHub, how GitHub supports collaboration, managing dependencies and tracking activity, as well as GitHub's effect on the development process. We sent our survey to 1,000 GitHub users and received 240 responses (24% response rate). We received several unexpected responses regarding the purpose of using GitHub. For example, respondents noted they used GitHub for purposes other than code hosting or collaborative development, such as for data storage, personal projects and class projects.

In an on-going project, we found that choosing which GitHub-based collaborative software engineering projects to study was not a trivial task. Frequently, projects were empty, had very few files or had been inactive for a long time. It was also common to find repositories where the only contributor was its owner.

These cases motivated our further analysis of the GitHub repository contents and of collaboration within GitHub, as discussed in sections 4.3 and 4.4. We then quantitatively and qualitatively analyzed the GitHub data to identify and measure the extent and frequency of perils. For the purposes of this paper, we define a "peril" as a characteristic of the data that can be retrieved from GitHub that can potentially threaten the validity of software engineering research that uses such data.

Our process was divided as follows:

1. ***Quantitative analysis of project metadata.*** We used the GHTorrent (Gousios and Spinellis, 2012) dataset made available in Jan 2014[3]. As described in section 1, the GHTorrent dataset is a comprehensive collection of GitHub repositories, their users, and their events (including commits, issues and pull requests). We also used the MSR'14 Mining Challenge Dataset(Baysal and Gousios, 2014) and cloned many repositories in GitHub in order to compare the GHTorrent data with current repositories.
2. ***Manual analysis of a 434-project sample.*** When quantitative analysis of metadata was not sufficient, we turned to in-depth manual analysis. We selected a random sample of 434 projects from the 3 million projects that exist in the GHTorrent dataset (cf. Peril I in section 4 for our definition of a project). This sample size provides a confidence level of 95% with a ±5% confidence interval.

4 Results

Through our mixed methods study, we identified thirteen perils that pose potential threats to validity for studies involving software projects hosted in GitHub (Table 1 summarizes them). In this section, we describe and provide supporting evidence for each peril, and include recommendations on how to avoid them.

4.1 Repositories are Part of Projects

Peril I: A repository is not necessarily a project

The typical pull request development model (as used by GitHub) is a newer method for collaborating in distributed software development **?**. With this model, the project's main repository is not writable by potential contributors. Instead, the contributors fork (clone) the repository and make their changes independent of each other. When a set of changes is ready to be submitted to the main repository, they create a pull request which specifies a local branch to be merged with a branch in the main repository. A member of the project's core team (a committer of the destination repository) is then responsible for inspecting the changes and pulling them into the project's master branch. If changes are considered unsatisfactory (e.g., as a result of a code review), more changes may be requested. In this case, contributors need to update their local branches with the new commits.

As a consequence of this popular development model, one can divide repositories into two types: base repositories (ones that are not forks) and forked repositories. The activity in forked repositories is recorded independently from

[3] `http://ghtorrent.org/downloads.html`

Table 1 Summary of the perils discovered in our study.

	Peril	**Description**
Project Related		
I	A repository is not necessarily a project	A project is typically part of a network of repositories: at least one of them will be designated as central, where code is expected to flow to and where the latest version of the code is to be found.
II	Most projects have low activity	Most projects have very few commits.
III	Most projects are inactive	Most projects do not have recent activity (only 13% of projects have been active in the last month).
IV	Many projects are not software development	A large portion of projects are not used for software development activities.
V	Most projects are personal	More than two thirds of projects (71.6% of repositories) have only have one committer: its owner.
VI	Many active projects do not use GitHub exclusively	Many active projects do not conduct all their software development activities in GitHub.
VII	Few projects use pull requests	Only a fraction of projects use pull requests. And of those that use them, their use is very skewed.
Pull Requests Related		
VIII	Merges only track successful code	If the commits in a pull request are reworked (in response to comments), GitHub records only the commits that are the result of the peer review, not the original commits.
IX	Many merged pull requests appear as non-merged	Only pull requests merged via the "Merge" button are marked as merged. But pull requests can also be merged via other methods, such as using `git` outside GitHub; in those cases, the pull-request will not appear as merged.
User Related		
X	Not all activity is due to registered users	The activity in GitHub repositories is sometimes due to non-users; in some cases, the activity of a user is not properly associated with her account.
XI	Only the user's public activity is visible	Approximately half of GitHub's registered users do not work in public repositories.
Github Related		
XII	GitHub's API does not expose all data	The GitHub API exposes either a subset of events or entities, or a subset of the information regarding the event or the entity.
XIII	Github is continuously evolving	GitHub continues to evolve and it has changed some features and provided new ones. Similarly, the projects evolve and are capable of changing their own history.

their associated base repositories. Until a commit is pulled into another repository, this commit appears only in the history of the recipient repository[4]. Therefore, measuring the activity of a repository independently of its forked repositories will ignore the non-merged activity of all of them as part of a single project.

For example, the Ruby on Rails project[5] has 8,327 forks (8,275 forks were made directly from its base repository, with the remainder being forks of forks). Of the 50k commits in the Rails repository, GHTorrent reports only 34k commits as having occurred in the Rails base repository (rails/rails), and the remaining 16k originating in its forks. However, 11k commits have been made in forks but have not been propagated to the base repository.

To properly account for all the activity of a software development team, in the rest of this paper we aggregate all the activity of the base repository and its forks. Thus we use the term **project** to refer to a base repository and its forks, and continue to use the term **repository** to denote a GitHub repository (either a base repository or a fork).

Of the 6.8M public repositories in GitHub, 3.0M (44%) are base repositories. Thus, these base repositories represent 3.0M different projects (only 0.6M of them have been forked at least once). For the base repositories with at least one fork, their number of forks is highly skewed: 80% have one fork only and 94% have at most 3 forks. However, there are some repositories that are heavily forked: 4,111 base repositories have been forked at least 100 times. The most forked repo is *octocat/Spoon-Knife* (22,865 forks), a GitHub administered repository for users to test how forking works.

It is important to highlight that many forks can operate independently from the rest of the project. For example, a fork could be used to develop customizations that are never intended to be contributed back into the main project. However, it is difficult to determine if a repository that has yet to contribute to the project will or will not contribute in the future.

Peril Avoidance Strategy: To analyze a project hosted on GitHub, consider the activity in both the base repository and all associated forked repositories.

4.2 On the Activity of Projects

Activity in GitHub is mostly reflected in commits—in all of GitHub, there are more than 20 times more commits than pull requests or issues. Thus, we can measure the activity of a project using two different proxies: by its number of commits and by the period in which its commits are made.

Peril II: Most projects have low activity

[4] GHTorrent associates a commit with the repository where it first sees it (table commits) and also links it to all repositories this commit has appeared into (table repo_commits)

[5] `http://rubyonrails.org` GitHub repository located at `https://github.com/rails/rails`.

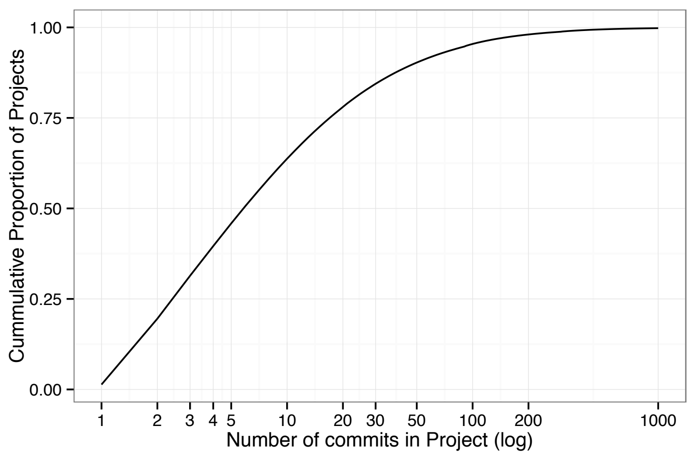

Fig. 1 Cumulative ratio of projects with a given number of commits (includes only projects with at least one commit). Most projects have very few commits. The median number of commits per project is 6 and 90% of projects have less than 50 commits.

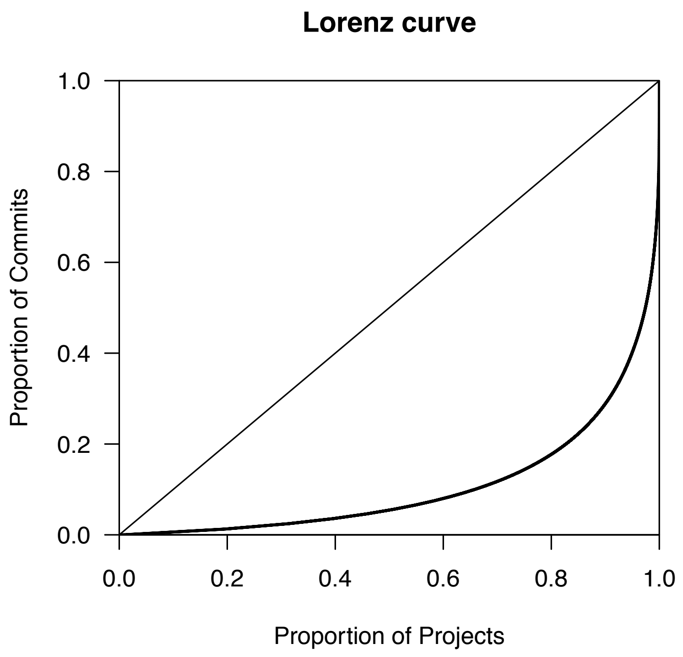

Fig. 2 Lorenz curve showing that a small number of projects account for most of the commits.

We measured the activity of commits per project—that is, the union of all the commits in all the repositories of a given project. Figure 1 shows the cumulative distribution (which is very skewed) with a median number of commits of only 6 and a maximum of 427,650 (these calculation do not include projects with zero commits–398,244 projects, 13.3%, had no commits).

Although there is a large number of projects with little activity, the most active projects account for the majority of commits in GitHub. This is shown in the Lorenz curve in Figure 2 that depicts the inequality of commits across the population of projects. The most active 2.5% of projects account for the same number of commits as the remaining 97.5% projects.

Peril Avoidance Strategy: Consider the number of recent commits on a project to select projects with an appropriate activity level. Avoid claims of generalization if your study considers only very active projects, as these are only a small set of those hosted on GitHub.

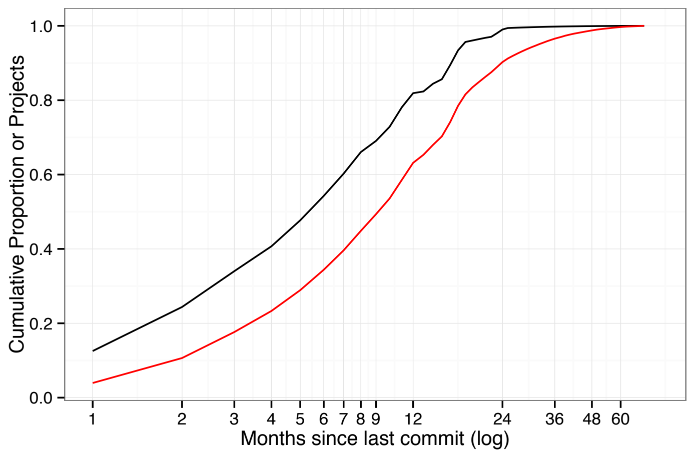

Fig. 3 Cumulative ratio of active projects during the last n months since Jan 9, 2014. The red line is the proportion of projects created during the last n months. Approximately 54% of projects have been active in the last six months. Only 12.5% of projects were active in the last month and 4% of them were created during that period.

Peril III: Most projects are inactive

If most of the projects have few commits, it is likely that they will also be inactive. Figure 3 shows the cumulative ratio of projects that have had activity during the last n months. For instance, in the last 6 months (since July 9, 2013), only 54% of the projects were active. However, many projects were created during this period (34% of all projects in GitHub). Of the 1,958,769 projects that were created before July 9, 2013, only 430,852 (22%) had at least one commit in the last 6 months.

We can also measure project activity by comparing the date its first repository was created in GitHub with the date of the project's last commit (as shown in Figure 4). In this regard, the median number of days a project is active is 9.9 days. 32% of projects were active for 1 day, suggesting that they are being used either for testing or for archival purposes. Only 38% were active for more than 1 month. However, many active projects continue to be active: 25% of projects have at least 100 days of activity.

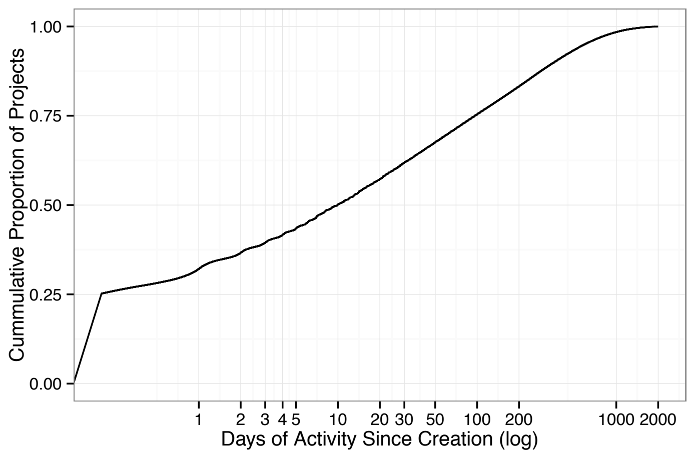

Fig. 4 Cumulative ratio of projects that had activity the last n days since their creation. The median number of days is 9.9, with 25% of projects at 100 or more days; only 32% had activity less than 1 day after being created.

Peril Avoidance Strategy: To identify active projects, consider the number of recent commits and pull requests.

4.3 On the Contents of Projects

Peril IV: Many projects are not software development

The answers to our survey indicated that GitHub is used for various purposes besides software development. 34 of our 240 respondents (14%) said they use GitHub repositories for experimentation, hosting their Websites, and for academic/class projects. About 10% of respondents use GitHub specifically for storage.

The purpose of a repository cannot be reliably and automatically identified from the project metadata. We used the 434 randomly selected repositories to determine if GitHub repositories are used for software development or other purposes; this sample provides a confidence level of 95% with a ±5% confidence interval. We reviewed the description of and files associated with each

Table 2 Number of repositories per type of use for the manual inspection. These categories are mutually exclusive.

Category of use	**Number of repositories**
Software development	275 (63.4%)
Experimental	53 (12.2%)
Storage	36 (8.3%)
Academic	31 (7.1%)
Web	25 (5.8%)
No longer accessible	11 (2.5%)
Empty	3 (0.7%)

repository and assigned an appropriate label to mark its contents, e.g, “software library” or “class project” using standard qualitative coding techniques (Corbin and Strauss, 2008). Open coding was used to identify labels for each repository. The open coding was performed by two individuals who each coded half of the repositories. After the open coding, the two coders agreed upon a set of labels and used axial coding to aggregate the labels to create exclusive categories of use. We defined the purpose of repositories as “Software development” if their contents were files that are used to build tools of any sort. This type of use included repositories of libraries, plugins, gems, frameworks, add ons, etc. “Experimental” was the class of repositories containing examples, demos, samples, test code and tutorial examples. Websites and blogs were classified under “Web”, and class and research projects under “Academic”. The “Storage” category included repositories that contained configuration files (including “.”’ files) or other documents and files for personal use, such as presentation slides, resumes and such. Repositories that gave an error (*404 This is not the repository you are looking for.*) were marked as “No longer accessible”. Repositories containing only a license file, a gitignore file, a README file, or no files at all were placed in the category “Empty”. Table 2 shows our categories and the distribution of the 434 repositories.

In particular, the “Web” category has become an important use of GitHub. GitHub allows its users to host Websites on its servers for free[6]. Repositories using this service typically include *github.io* or *github.com* in their name. There are 73,745 projects with such names, indicating the popularity of this free service.

Peril Avoidance Strategy: When trying to identify which software development projects to analyze, do not rely just on the types of files within the repositories, but also also review descriptions and README files to ensure the projects fit the research needs.

4.4 On the Users Involved with Projects

Peril V: Most projects are personal

[6] See `http://pages.github.com/` for details.

Our survey asked respondents if they used GitHub primarily for collaboration with others or for personal use. 90 out of 240 respondents (38%) answered that they used GitHub mainly for their own projects and not with the intention of collaborating with others. This response was a motivating factor to look into how much collaboration and social interaction is taking place in GitHub projects.

In `git`, a commit records both its author (who wrote the patch) and its committer (who committed the patch to the repository). The committer is the person who has write access to the repository. In GitHub, only 2.9% of commits have an author who is not the committer. We can evaluate if a project is personal by counting the number of different committers in all the repositories of the project.

The number of committers per project is very skewed: 67% of projects have only 1 committer, 87% have 2 or less, and 93% have 3 or less. As expected, repositories have fewer committers than projects: 72% have 1 committer, 91% have 2 or less, and 95% have 3 or less. The proportions are the same for numbers of authors. The number of committers in our manual sample is similar: 65% hand only one committer, 83% two or less, and 90% three or less.

These results indicate that even though GitHub is targeted towards social coding, most of the projects it hosts are used by one person only. It is very likely that a large proportion of projects with only one committer are for experimental or storage purposes.

Peril Avoidance Strategy: To avoid personal projects, consider the number of committers.

4.5 On the Use of Non-GitHub Infrastructure

Peril VI: Many active projects do not use GitHub exclusively

A difficult question to answer is whether the data in GitHub represents most (if not all) the visible activity of a development project. In other words, do projects in GitHub use other forms of collaboration?

There were indications in the survey responses pointing towards project activity talking place outside GitHub. As one of the respondents put it:

> *"Any serious project would have to have some separate infrastructure - mailing lists, forums, irc channels and their archives, build farms, etc. [...] Thus, while GitHub and all other project hosts are used for collaboration, they are not and cannot be a complete solution."*

This motivated us to look into whether repositories host project code and other content on GitHub, but perform development and collaboration activities elsewhere.

There are several ways we could evaluate this. One of them is to determine if all the committers and authors are users in GitHub. If a commit is made by someone who is not a GitHub user, then GitHub records an email address as its committer rather than a GitHub user (see *Peril X Not all activity is due to registered users*). In GitHub, 23% of committers or authors of a commit are not GitHub users. The likely reason for this result is that some `git` operations from non-users have been merged outside GitHub and it is exacerbated by mirrors set up to track activity in repositories outside GitHub.

Mirrors are replicas of the code hosted in another repository. In some cases, a mirror project clearly indicates that GitHub is not to be used for submission of code. For example, the project *postgres-xc/postgres-xc* states in its description *"Mirror of the official Postgres-XC GIT repository. Note that this is just a *mirror* - we don't accept pull requests on github..."*. Nonetheless, this project has 14 different forks.

We identified many repositories that are mirrors—GitHub officially maintains 91 mirrors of many popular projects[7]. Typically, the description of a repository states if it is a mirror. For example, the description of repository *abishekk92/voipmonitor* reads "A mirror of the SVN repo at `https://voipmonitor.svn.sourceforge.net/...`". Descriptions can also indicate whether the mirror is automatic and note its frequency of update (e.g., "Mirror of official clang git repository located at http://llvm.org/git/clang. Updated hourly.").

The case-insensitive regular expression `mirror of .*repo|git mirror of` finds 1,739 projects (12,709 repositories) as mirrors of repositories outside GitHub. The median number of commits is 52. Some of there repositories had a lot of activity: 78 had more than 1,000 commits (1.4% of all repos with at least 1,000 commits). We examined 100 of these repositories and found that all of them were external mirrors. We identified many mirrors from SourceForge repositories and Bitbucket (a competing `git` repository hosting service)—these results are summarized in Table 3.

The implications of these results is that part of the development of a project happens in GitHub, but not necessarily all.

Table 3 Repositories hosted on GitHub labeled as mirrors. GitHub hosts mirrors from many sources, including SourceForge and Bitbucket. The bottom section shows subsets of the top section. Regular expressions are case insensitive.

Set	Used regular expresion	Projets	Repos
Mirror of	`mirror of .*repo|git mirror of`	1,851	12,709
Subsets			
Located on Sourceforge	`sourceforge|sf\.net`	117	511
Located on Bitbucket	`bitbucket`	91	249
From subversion repos	`\W(svn|subversion)\W`	622	4966
From mercurial repos	`\W(mercurial|hg)\W`	113	590
From CVS repos	`\Wcvs\W`	55	212

The development within a mirror in GitHub implies that some members of a project are using GitHub for one of two purposes. One purpose is to develop

[7] https://github.com/mirrors

their work and later submit it to the external repository. For example, the project *Linux-Samsung* located at *kgene/linux-samsung* (which, according to GitHub, has no forks and is not a fork itself) regularly contributes commits to the Linux kernel (we have observed 123 commits in Linus Torvald's repository that originated here[8]). The second purpose is to develop customizations of the original project for a different purpose, independent of the original development team. In this category, we find multiple repositories that contain variants of the kernel, such as *2.6.35 Kernel for Samsung Galaxy S series Phones* or *Kernel 2.6.35.7 modified for Dropad A8T and similar.*

Interestingly, some mirrors are from repositories that use other version control systems, such as Mercurial, Subversion or CVS. This implies that, in some cases, contributors prefer `git` over these other version control systems to do their daily work, but this needs further research to be confirmed. Similarly, many projects use their own defect tracking systems to handle issues. For example, Mozilla's Gaia (`mozilla-b2g/gaia`), one of the most active projects in GitHub, has disabled issue tracking in GitHub and expects users to file issues through `bugzilla.mozilla.org`.

We conducted a small survey that asked respondents to tell us whether they used GitHub's tools or an external toolset for specific tasks, such as opening and merging pull requests, tracking issues, or for communication. We sent an additional questionnaire to 100 GitHub users via email and received 27 responses (27% response rate). Even though 52% said they use GitHub to open pull requests and 60% said they use the site to accept and merge code changes, only 24% said they use GitHub for code reviews. 32% said they use an external tool for reviews. This further validates that all software development activities do not occur within GitHub itself for many projects.

Peril Avoidance Strategy: Avoid projects that have a high number of committers who are not registered GitHub users and projects with descriptions that explicitly state they are mirrors.

4.6 On Pull Requests

Promise II: GitHub provides a valuable source of data for the study of code reviews in the form of pull requests and the commits they reference

GitHub made the "Fork & Pull" development model popular, but pull requests are not unique to GitHub. In fact, `git` includes the `git-request-pull` utility which provides the same functionality at the command line. GitHub and other code hosting sites improved this process significantly by integrating code reviews, discussions and issues, thus effectively lowering the entry barrier for casual contributions. Forking and pull requests create a new development model where changes are pushed to the project maintainers and go through code review by the community before being integrated.

[8] We currently track all sources of commits in the Linux kernel: `hydraladder.turingmachine.org`

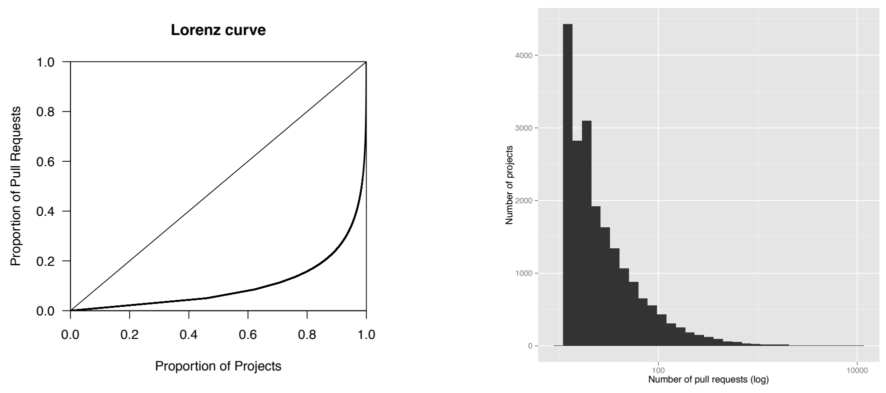

Fig. 5 Lorenz curve for the number of pull requests per project (left) and the corresponding histogram (right). The top 1.6% of projects use 50% of the total pull requests. These plots only include projects with at least one pull request.

Peril VII: Few projects use pull requests

Across GitHub, the use of pull requests is not very widespread. Pull requests are only useful between developers, and therefore, are non-existent in personal projects (67% of projects, see *Peril V Most projects are personal*). Of the 2.6 million GitHub projects that represent actual collaborative projects (at least 2 committers), only 268,853 (10%) used the pull request model at least once; it is likely that the remaining 2.4M projects are using a shared repository model exclusively (with no incoming pull requests) where all developers are granted commit access. Moreover, the distribution of pull requests among projects is highly skewed, as can be seen in Figure 5. The median number of pull requests per project is 2 (44.7% of projects have only 1 and 95% have 25 or less).

Nonetheless, there are projects that received more than 5,000 pull requests in 2013 alone, such as the Gaia phone application framework and the Homebrew package manager. In fact, a significant number of projects (∼1700) received more than 100 pull requests in 2013. These projects can create a sample big enough to deliver statistically significant results for many research questions.

Peril Avoidance Strategy: When researching the code review process on GitHub, consider the number of pull requests before selecting a project. Personal projects will rarely contain pull requests.

4.6.1 Pull Requests as a Code Review Mechanism

A GitHub pull request contains a branch (local or in another repository) from which a core team member should pull commits. GitHub automatically discovers the commits to be merged and presents them in the pull request. By default, pull requests are submitted to the destination repository for review. There are two types of review comments:

– Discussion: Comments on the overall contents of the pull request. Interested parties engage in technical discussion regarding the suitability of the pull request as a whole.
– Code Review: Comments on specific sections of the code. The reviewer makes notes on the commit diff, usually of a technical nature to pinpoint potential improvements.

Any GitHub user can participate in both types of review. As a result of the inspection, pull requests can be updated with new commits or the pull request can be rejected—either as redundant, uninteresting or duplicate. The exact reason a pull request is rejected is not recorded, but can often be inferred from the comments.

With an update, the contributor creates new commits in the forked repository and, after the changes are pushed to the branch to be merged, GitHub automatically updates the commits in the pull request. The code review can then be repeated on the refreshed commits. In our dataset of 434 projects, 17% of the pull requests received an update after a comment (discussion or code review). Care must be applied when interpreting this result as many comments, especially in the discussion section, are merely expressions of gratitude for the contributor's work rather than a proper code review.

The discussion around a pull request is usually brief: 80% of the pull requests have less than 3 comments (both code review and discussion). Moreover, the number of participants in the code review ranges between 0 and 19, with 80% of the pull requests having less than 2 participants. The number of commits examined per peer review is less than 4 in 80% of the pull requests. The numbers are comparable with other work on code review (Rigby et al, 2008; Rigby and Bird, 2013; Bacchelli and Bird, 2013) which suggests that the peer review process may have more fundamental underpinnings yet to be explored. Therefore, GitHub data may be a very good source of quantitative data for peer review due to homogenization across various project repositories (provided the following shortcomings are taken into consideration).

It is important to note that code reviews in pull requests are in many cases implicit and therefore not observable. Many pull requests that were merged received no comments (46% in our 434-project sample). It is probably safe to assume that the developer that performed the merge did inspect the pull request before merging it. Thus, a code review occurred, but there is no information about it except the fact that the code was merged (it is unlikely that a project will have a policy to accept all pull requests without review).

Peril VIII: Merges only track successful code

Another shortcoming of using GitHub data for peer review research is the fact that the set of commits that were reviewed might not be readily observable—and might require further processing to recover them. Commonly, projects require a commit squash (merging all different commits into a single one) before the set of commits is merged with the main repository. While GitHub does record the intermediate commits, it does not report them through its API as part of the pull request. Moreover, the original commits are deleted if the source repository is deleted. This means that at the time of analysis, the researcher can only observe the latest commit, which is the outcome of the code review process.

Peril Avoidance Strategy: To analyze the full set of commits involved in a code review, do not rely on the commits reported by GitHub.

Peril IX: Many merged pull requests appear as non-merged

When the code review is finished and a pull request is deemed satisfactory, the pull request can be merged. The versatility of `git` and GitHub enables at least three merging strategies:

– Through GitHub facilities, using the "Merge" button.
– Using `git`, by merging the main repository branch and the pull request branch. A variation of this merge strategy is cherry-picking, where only specific commits from the pull request branch are merged into the main branch.
– By creating a textual patch between the pull request and main repository branches and applying it to the master branch. This is also known as commit squashing.

The merge strategies presented above differ in the amount of history (commit order) and authorship information preserved. Specifically, merging through either `git` or GitHub preserves full historical information—except in the case of cherry-picking where only authorship is preserved. A patch-based merge does not maintain authorship or history.

Moreover, GitHub can only detect and report merges happening through its pull request merge facilities. Therefore, if a project's policy is to only merge using `git`, all pull requests will be recorded as unmerged in GitHub. In practice, however, most projects use a combination of GitHub and `git` merge strategies.

To streamline the closing of pull requests and issues, GitHub provides a way to close them via the contents of the log of a commit. For example, if a commit log contains the string *Fixes #321* and 321 is a pull request or an issue, then this pull request or issue is closed. *Fixes* is one of nine keywords that can be used[9]. For example, the project *homebrew/homebrew* has had 13,164 pull requests opened, 12,966 closed, but only 129 merged. However, its logs show that 6,947 pull requests (48% of total) and 2,013 issues (19%) have been closed from commit logs. This shows that, at least in some projects, one cannot rely on GitHub's *Merged* attribute of a pull request.

To identify merged pull requests that are merged outside GitHub, we have developed a set of heuristics based on conventions advocated by GitHub. The most important are presented below (for a full description and evaluation of these heuristics, see (Gousios et al, 2014)).

[9] For the entire list visit `https://help.github.com/articles/closing-issues-via-commit-messages`.

H_1 At least one of the commits in the pull request appears in the target project's master branch.

H_2 A commit closes the pull request using its log (e.g., if the log of the commit includes one of the closing keywords, see above) and that commit appears in the project's master branch. This means that the pull request commits were squashed onto one commit and this commit was merged.

H_3 One of the last three (in order of appearance) discussion comments contain a commit unique identifier—this commit appears in the project's master branch and the corresponding comment can be matched by the following regular expression:

```
(?:merg|appl|pull|push|integrat)(?:ing|i?ed)
```

H_4 The latest comment prior to closing the pull request matches the regular expression noted above.

Across GitHub, 1,145,099 of 2,552,868 (44%) pull requests are reported as merged. In the 434-project sample, only 37% of the pull requests were merged using GitHub facilities. By applying the heuristics presented above, an extra 42% (H_1: 32%, H_2: 1%, H_3: 5%, H_4: 4%) of pull requests are identified as merged, while 19% cannot be classified. In other work (Gousios et al, 2014), we used a carefully selected sample of 297 projects that heavily relied on pull requests: 65% of the pull requests were merged using GitHub facilities, while the heuristics identified another 19% (H_1: 7%, H_2: 1%, H_3: 3%, H_4: 7%) as merged. In another dataset (Gousios and Zaidman, 2014b) containing almost 1000 projects that use pull requests, 58% of the pull requests were merged using GitHub's facilities while 18% are identified as unmerged. The remaining 24% are identified as merged using the heuristics (H_1: 11%, H_2: 3%, H_3: 3%, H_4: 7%) .

The heuristics proposed above are not complete (i.e., they may not identify all merged pull requests) nor sound (i.e., they may lead to false positives, especially H_4). In other work (Gousios et al, 2014), we manually inspected 350 pull requests that were not identified as merged and found that 65 of them were actually merged. This means the actual percentage of merged pull requests may be even higher. The fact remains, however, that only a fraction of merges are reported through GitHub, but heuristics can improve merge detection, in some cases dramatically.

Peril Avoidance Strategy: Do not rely on GitHub's merge status, but consider using heuristics (like the ones described above) to improve merge detection when analyzing merged pull requests.

4.6.2 Pull Requests as an Issue Resolution Mechanism

Promise III: The interlinking of developers, pull requests, issues and commits provides a comprehensive view of software development activities

Issues and pull requests are fused together on GitHub: for each opened pull request, an issue is opened automatically. Commits can also be attached to issues to convert them to pull requests (albeit with external tools). The issue part of the pull request is used to keep track of any discussion comments. Developers are encouraged to reference issues or pull requests in commit messages or in issue comments, while GitHub automatically extracts such references and presents them as part of the discussion flow. Moreover, both issues and pull requests can be linked to repository-specific milestones, helping projects track progress.

The fact that issues and pull requests are so tightly integrated opens a window of opportunity for detailed studies of developer activity. For example, a researcher can track the resolution of an issue from the reporting phase, through source code modifications, the code review and the final integration of the fix. As user actions always affect issues and pull requests, one could also investigate the formation of user clusters across specific types of activities, which would reveal emergent user organizations (teams or hierarchies). In addition, the interlinking of issues, pull requests and commits creates an intricate web of actions that could be analyzed using social network techniques to discover interesting collaboration patterns.

Despite the wealth of interlinked data, there are two shortcomings. First, repository mining for issue tracking repositories is greatly enhanced if records are consistent across projects. GitHub's issue tracker only requires a textual description to open an issue. Issue property annotations (e.g., affected versions, severity levels) are delegated to repository-specific labels. This means that issue characteristics cannot be examined uniformly across projects. Second, across GitHub, only a small fraction (12%) of repositories that where active in 2013 use both pull requests and issues. Many interesting repositories, especially those that migrated to GitHub, have an external issue database (see *Peril IV Many active projects do not use GitHub exclusively*).

4.7 On Users

Peril X: Not all activity is due to registered users

GitHub is a service built around `git`. A team of developers who use `git` can choose to use GitHub for all or some of their development activities. GitHub enables teams to import their `git` repositories into GitHub, even if some members of the development team are not GitHub users. In some cases, such as with "mirrors", it is possible that no one on the development team is a registered GitHub user. This implies that some activities recorded in GitHub are not performed by its registered users.

GitHub allows users to associate one or more email addresses with their account (no two users can share the same email address). When GitHub receives a commit into a repository via a push, it uses the email address of the committer and the author field of the commit to associate the commit with a corresponding GitHub user. If the email address is not registered to a user account, the commit is not linked to the account. For example, 15 repositories belonging to *Kevin Incorvia* (username *incorvia*) contain commits linked to the email address *Kevin Incorvia <incorvia@Kevins-MacBook-Air.local>*, but they are not associated to user *incorvia* because the email address is not associated with that user. Furthermore, the email address is more likely a computer username rather than an actual email address. We could speculate that this is due to using a `git` client, which pulled the GitHub user's username and the host name of their machine to combine it into an email address, rather than asking them to enter an email address, as is the case of using the command line. In any case, if one were to ask for the activity associated with user *incorvia*, these commits would not be included. A similar case is the email address being empty or invalid. For example, the repository *TrinityCore/TrinityCore* contains commits by the email address *megamage <none@none>*. These commits are not associated with any user—GitHub's interface does not even show a committer section while displaying the details of the commit and its API shows *null* as the author/committer). The impact of this association of emails to commits is four-fold.

- **Not all committers or authors of commits are registered GitHub users.** By December 2014, we had identified 2.5M registered users and 0.6M email addresses that could not be associated to registered users (we refer to these email addresses as non-registered users); 84.4% of commits (65.2M) were performed by registered users and 15.6% (12.1M) by non-registered users. Pull requests, issues and their comments can only be made by registered users.
- **A small number of users have commits that predate the creation of their GitHub user account (1.5%, 33,227)**. For example, Linus Torvalds joined GitHub on September 3, 2011, but has commits associated with his user account as early as September 4, 2007.
- **A committer can make a commit appear as coming from another user by using one of the other user's email addresses.** For example, the commit 042343a09967445753b174b0b05c6ef3cfcf7f93 in the repository *aaronraimist/public* shows *Aaron Raimist <torvalds@linux-foundation.org>* as its author and committer, yet the commit is associated with *Linus Torvalds* and not with *Aaron Raimist* (Aaron is also the owner of the repository where the commit was found). This issue is probably rare and is difficult to identify.
- **It might be necessary to perform email unification to fully identify the activity of each user.** While GitHub allows a user to have multiple addresses associated with their account, the user must associate all their addresses. However, not all users have registered all the email addresses they have used. For example, Linus Torvalds has commits in GitHub with 10 email addresses (from the Linux Foundation and the Open Source Development Labs) that are not associated with his GitHub account. They have been used in 17,460 of his commits in GitHub, while his GitHub account has 19,780 commits associated with it. In other words, 47% of Torvalds' commits in GitHub are not associated with his user account. To

quantify this effect empirically, we devised the following experiment. For each project, identify persons who commit with two different email addresses (the same name, but two different email addresses) and their name contains at least two words. The assumption is that for each project there is only one person with a given firstname-lastname combination. In other words:

- Select committers who have a name with at least one space in between. This step selects committers with at least two words in their name (e.g. *Linus Torvalds*) and avoids matching people who share the same firstname or lastname (e.g. *David*).
- For each of these names, count how many email addresses they used in a project. If the user is registered, we use their preferred email address for the commit. If the user is not registered, we use the email address in the commit.

Note that this method is likely to underestimate duplicated emails per user since there may be emails that lack a name, or their name may only contain one word, or a person uses different ways to write their name (e.g., *J. Smith* and *John Smith*).
We found that only 30.8% (664,850) of registered GitHub users have two or more words in their name, and 17% of them (90,828 users) have at least one email address that is not associated with their username in the same project (median of 2). These email addresses have committed 2.09 million commits (2.7% of all commits). This number, however, corresponds to 22.1% of commits by non-registered committers. In other words, it is possible to associate approximately 1/4 of commits by non-registered committers to their corresponding GitHub user. This effect seems to be small, but it shows that a significant proportion of users have identities that have not been unified.

Peril Avoidance Strategy: For empirical studies that need to map activity to specific users, use heuristics for email unification to improve the validity of the results.

Peril XI: Only the user's public activity is visible

When we take a closer look at the activity of registered users, we notice substantial inactivity. Before concluding that GitHub users appear to be generally inactive, however, we need to keep in mind that we can only see public activity (actions that take place in public repositories).

Let us focus on the commit as the basic unit of activity on GitHub. 97% of the time, the committer and the author are the same person. Hence, for this analysis we consider them equivalent and look only at the committer field to estimate commit activity.

We found that out of the registered users on GitHub, 53.2% do not have a single public commit. This population can be further divided into two parts: 30% of the registered users do not have a public repository either, while the remaining 23.2% have repositories but no public commits. These repositories fall into two categories: empty repositories (e.g., used for testing) and forks that have no activity.

The remaining 46.8% of registered users (1.04 million) have at least one commit. As shown in Figure 6, the distribution of commits per user is highly skewed: the median is 10 commits with an average of 62.4 commits. The inequality in the number of commits per registered users is substantial (see Figure 7): 50% of the commits have been performed by 3.2% of registered users, and 25% by 0.6% of them. This is mainly due to the fact that some registered users appear overly active with a very high number of commits.

However, considering only commits as a measure of activity excludes users who are active in other ways. 24.4% of registered users who do not have any commits have submitted issues or participated in discussions around issues or pull requests. This shows that there is a subset of GitHub users who do not publicly develop code (they could have private repositories), but are actively contributing to GitHub repositories by identifying bugs, submitting new feature requests, reviewing code or simply providing guidance to developers.

Peril Avoidance Strategy: This peril is unavoidable when using data from public Websites—acknowledging this partial view in the discussion of results and replicating a study in other contexts can help reduce its impact.

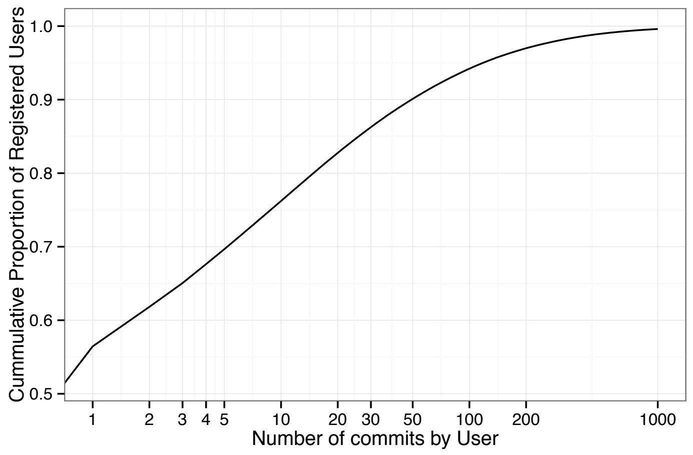

Fig. 6 Cumulative ratio of registered users that have n commits. 50% of users (0.52 million) have less than 10 commits and only 10% (0.1 million) have more than 50 commits.

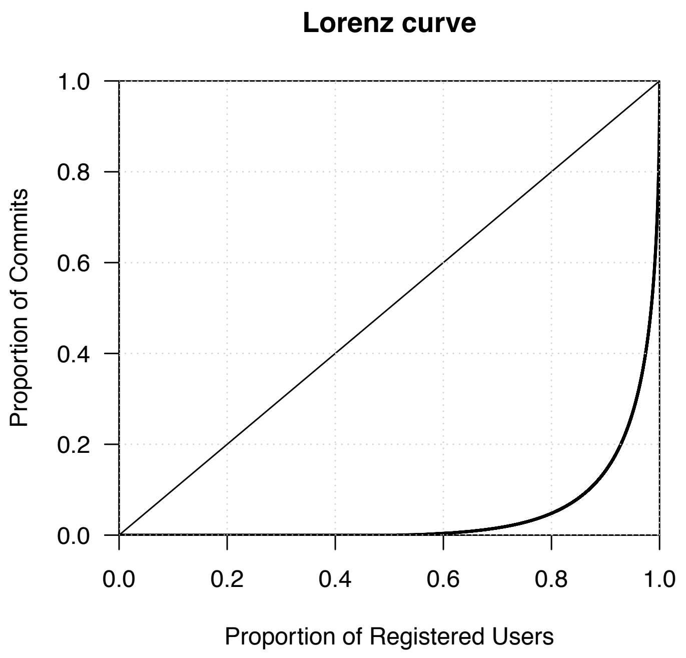

Fig. 7 Lorenz curve showing that most commits by registered users are made by a small proportion of them. E.g., 50% of the commits by registered users have been performed by 3.2% of them.

4.8 GitHub is an Evolving Entity

GitHub is not operating as an archive of software development activities for research purposes; its goal is to provide "powerful collaboration, code review and code management for open source and private projects."

Peril XII: GitHub's API does not expose all data

While the repositories hosted in GitHub are continuously evolving, GitHub reports only their current state (via its API). It does not report the historical events that shape the current state of any repository. This results in several challenges for researchers:

– **GitHub does not provide an API to retrieve all events.** GitHub has created APIs to list many of its entities (such as users, repositories and commits) and events (such as opening or closing issues, or pull requests). However, it does not make them all available. For example, GitHub does not expose pushes to a repository, the creation of releases, clone operations or when a repository is made public. Some of these events are available in the events API of a repository, but this API has the limitation of only listing the last 300 events.

- **Not all events are reported with a timestamp.** In particular, the APIs for subscriptions and “stars” do not return the time when such actions were initiated.
- **Tracking renamed entities.** A renamed repository keeps all its information under the new name (including its forks). GitHub will redirect the URLs of renamed repositories, but it will not do it for API requests. For example, the repository *anders9898/jekyll* was later renamed to *anders9898/zzz*. The URL `https://github.com/anders9898/jekyll` redirects to `https://github.com/anders9898/zzz`, but the API request `https://api.github.com/repos/anders9898/jekyll` returns “Not Found”. In the case of users, there is no way to know that a user has been renamed. In this case, both the URL and the API requests for the old name will fail.
- **Deleted entities.** When a repository is deleted, all of its events and metadata are lost, but the network of repositories that were forked from it remain untouched; one of its forks will be chosen as the “root” of the rest of the forks. From this point on, GitHub will report the deleted repository as “Not found”. Similarly, when commits are deleted, GitHub has no mechanism to inform that certain commits were deleted from a repository. Finally, once a user is deleted, all information regarding them (including the fact that they were a GitHub user) is lost.
- **Making a repository private causes it to appear as if it were deleted.** Similar to when a repository is deleted, GitHub will report such a repository as “Not found”. In this case, its forks will remain public and one of them is chosen as their root.
- **Rebased commits.** When commits are rebased (one or more commits changes its metadata, or are modified and/or combined into new commits) the old commits disappear from the repository and are replaced by the new commits. GitHub’s events API does not document commits that are removed or modified in a push; it only lists the commits that are added and the head of the branch before and after the commit. Any request for the old commits in the repository will result in a “Not found” message. The commits that are deleted or rebased in one repository might still exist in another if they have already been propagated there.
- **Propagation of commits.** GitHub tracks the movement of commits between repositories only when commits are merged using a pull request and those commits have not been rebased or deleted after the merge. In this case, the merged pull request will document what commits were merged, including the source and destination repositories. If the merge was performed outside GitHub, then GitHub has no means of knowing what the true source of the commits was. For example, assume Sally creates a fork F of repository A and then clones it to her computer. Next, she “pulls” changes from another fork G and then pushes the commits to her GitHub fork F. Under this scenario, GitHub has no means of knowing that the new commits in her repository came from G. If, on the other hand, there was a pull request from G to F and Sally merges it, then GitHub will know that those commits were moved from G to F.

Peril Avoidance Strategy: Obtaining data from one of the services which archives data from the GitHub API (like GHTorrent) can help avoid this peril. However, researchers should be aware that such services contain their own assumptions regarding the collected data.

Peril XIII: Github is continuously evolving

Over time, GitHub has changed some of its features and interface. For example, GitHub's "watch" feature was originally intended to be used by those who wanted to receive notifications regarding activity (commits, pull requests and issues) for any repository of their choosing. In August 2012, GitHub decided to improve their notification system (Neath, 2012). The first change was the introduction of "starring". "Starring" a repository is equivalent to bookmarking it. Any previously "watched" repository became a "starred" repository, and the old notification system surrounding "watchers" changed to an opt-out subscription of events notification system (a person with commit privileges to a repository automatically "watches" the repository). As a way to maintain backwards compatibility with external applications that used this information, GitHub currently returns the list of "starred" projects under its *Watched* API (e.g., /users/:user/watched). This introduces two potential problems for researchers. First, the meaning of "watchers" is different before and after August 2012. Second, there is a dissonance between GitHub's interface and its API: "Watchers" are returned using the *Subscriptions* API, and "stars" (those who have "starred" a repository) are returned via the *Watched* API.

More recently (November 2014), GitHub *silently* disabled the ability to retrieve repository collaborators for a specific repository. The only way to retrieve this information now is to query and keep track of the live event stream for a particular repository or sets or projects.

Both the above changes had a direct impact on the API. There are also frequent changes to the GitHub interface that do not leave an API footprint, yet they have the potential of changing user behavior. As an example, the issue tracking system in GitHub was improved significantly on July 28, 2014[10]. The interface changes related to improving search and filtering, showing a timeline of issue-related activities (such as assigning labels, changing issue names and adding comments) and improving the editing of milestones and labels for issues. Although the user data created and stored was not affected, researchers should still keep track of when the GitHub interface changed and what the improvements were. Capturing and measuring the actual impact of interface changes on user behavior would be a separate research undertaking as it is beyond the scope of this paper. However, we want to keep researchers alert that patterns seen in the data could be explained by accounting for changes in the interface, reflecting changes in user behavior, even if they were not captured in the API.

10

urlhttps://github.com/blog/1866-the-new-github-issues

Table 4 Percentage of projects susceptible to more than one perils. The table is read as: "From the repositories that are susceptible to peril x (column) Y% are also susceptible to peril z (row)". We only include perils whose effect can be quantified. The table is valid for January 2014 (total repository population: 5,397,054).

Peril	Number of repos	P I	P II	P III	P V	P VI	P VII	P X
P I	700,314	—	100	100	100	100	100	100
P II	156,328	22	—	24	12	10	19	14
P III	587,453	83	93	—	78	0	70	70
P V	117,232	20	12	19	—	28	25	0
P VI	58,023	8	3	0	11	—	8	15
P VII	464,426	66	70	71	59	65	—	52
P X	171,780	25	15	20	42	42	37	—

Peril	Quantification method
P I	Repositories that have forks.
P II	Of the repositories in P1, those that have less than 6 commits.
P III	Of the repositories in P1, those with no activity (commit, pull request, issue) in Dec 2013.
P V	Of the repositories in P1, those where only one user has commit access.
P VI	Of the repositories that were active in Dec 2013, those that only had commit activity.
P VII	Of the repositories in P1, those that never received a pull request (any time).
P X	Of the repositories in P1, those featuring commits belonging to non-users (fake users).

Peril Avoidance Strategy: Understand how both the GitHub API and the Website have evolved over time. Changes to the Website are often posted to the GitHub blog[11], but this is not guaranteed.

4.9 Relationship between perils

It is possible for one project to be subject to more than one perils. To calculate the extend this can happen in our dataset, we calculated the pairwise appearance of the perils in our dataset. More formally, for each peril P_A in the set of perils $P = \{P_1, ..., P_10\}$, we calculated a list of projects L_{P_A} which this peril may affect. Then, we examined whether each peril T_B in the set $T = P \setminus \{P_A\}$ also affects the projects in L_{P_A} and therefore came up with a second project list $L_{P_A}^{T_B}$. The ratio of projects in L_{P_A} that are affected by both perils P_A and T_B is $|L_{P_A} \cap L_{P_A}^{T_B}|/|L_{P_A}^{T_B}|$. We only did this for perils that could be quantified given our dataset. The results along with the quantification method can be seen in Table 4.

Overall, we can see that there is a significant, but not absolute, overlap among the perils as they manifest in the projects in our dataset. Peril III (most projects are inactive) seems to be strongly related with most other perils, while Peril VII (most projects do not receive a pull request) has significant overlap with perils related to project activity. Peril I is naturally perfectly correlated with all other perils as the population samples were drawn after

[11] `https://github.com/blog/category/ship`

Peril I was evaluated. In light of this, it is interesting to note that 83% of the forked repositories (potential projects) were inactive in the month prior to our dataset snapshot while 66% never received a pull request (therefore, the forks did not contribute back).

What is perhaps more interesting about peril overlaps is that when read in reverse, they provide us with simple guidelines to guide project selection for research. While several rules can be extracted, we believe that the following should be part of any effort wanting to examine healthy projects.

1. Choose repositories that have forks.
2. From those remaining after Step 1, exclude repositories that were inactive for a predefined period prior to experimentation.
3. From those remaining after Step 2, exclude repositories that have never received a pull request.

The remaining repositories can then be filtered according to further criteria, for example number of stars (popularity) or programming language.

5 An Analysis of the MSR 2014 Mining Challenge

In the MSR 2014 Mining Challenge (Baysal and Gousios, 2014), researchers were given a subset of the GHTorrent dataset to analyze and derive new insights. The competition resulted in nine accepted papers. In view of the perils presented, we analyzed the dataset and accepted papers to determine if any of our perils might have posed potential threats to validity to the results presented in these papers.

5.1 The Dataset

The organizers of the Mining Challenge decided that the entire GHTorrent dataset was too large. Instead, 90 repositories were selected as follows: for each of the top 10 programming languages (including Javascript, Java and other popular languages), the top 10 most active repositories in terms of pull requests processed in 2013 (up to September 2013) where initially selected. The original selection was then hand-cleaned to remove repositories that where not software development ones.

Below we discuss how the identified perils could have an impact on insights derived from this dataset.

Peril I A repository is not necessarily a project: The data contains 90 projects, but unfortunately, the schema of the dataset refers to a repository as a "project" and it does not include an entity for "project". Projects must be inferred by recursively traversing the $forked\_from$ field of the "projects" table to identify the project a repository belongs to. However, one repository could not be linked to its project (*xphere-forks/symfony*).

Peril II Most projects have low activity: There are 3 projects with less than 100 commits, while 3 have more than 40,000 commits; 10 repositories account for 50% of the commits.

Peril III Most projects are inactive: The impact of this peril is small: only 4 repositories were inactive in the last 6 months. In contrast, 65 repositories had been active in the last week and 71 in the 2 weeks before.

Peril IV Many projects are not software development: The impact of this peril is also small: one repository was a personal Website (*vinc/vinc.cc*), while another one is a book on R programming (*mavam/stat-cookbook*). Another is a collection of icons (*FontAwesome/Font-Awesome*).

Peril V Most projects are personal: One of the projects only had one committer (*vinc/vinc.cc*), and one had three committers (*mavam/stat-cookbook*). Again, the impact of this peril is small.

Peril VI Many active projects do not use GitHub exclusively: *jquery/jquery*, *mono/mono*, *ServiceStack/ServiceStack*, *django/django*, *clojure/clojure* do not use GitHub for issues. For example, Clojure uses Jira—this is where it suggests non-regular contributors should submit patches instead of GitHub. Jquery hosts its own bug tracking system at `bugs.jquery.com`. Mono uses bugzilla `www.mono-project.com/Bugs`. At least one repository is a mirror (*TTimo/doom3.gpl*) with incomplete development history; it was imported from a release of the game.

Peril VII Few projects use pull requests: In this dataset, the majority of the projects use pull requests: 88 of the 90 repositories. The median number of pull requests per project is 393. However, 3 repositories account for 34% of the pull requests (*mxcl/homebrew*, *rails/ralis* and *symfony/symfony*), and 7 repositories account for 50%.

Peril VIII Merges only track successful code: This is an overarching peril inherent in GitHub data due to the way GitHub reports merged pull requests and the commits they contain.

Peril IX Many merged pull requests appear as non-merged: Some projects do not close many of their pull requests using the GitHub "Merge" button. Instead, they do it via commits in their local repositories. One project, *mxcl/homebrew*, poses an important threat to validity if we assume pull requests that are not marked-as-merged were not actually merged. This project is the one with the most pull requests (it accounts for 17% of pull requests in the dataset) and only 0.9% them are marked as "merged". However, as part of their development process they close pull requests via the log of a commit. We found (Gousios and Zaidman, 2014a) that 52% of pull requests had actually been merged (6,753 pull requests in the MSR dataset were actually merged). *django/django* also does not always close the pull request via the "Merge" button. In that case, we found that 819 pull requests had been merged (41% more, for a total of 63%, compared to 23% found in the MSR dataset). In *Bukkit/CraftBukkit*, 23.8% of commits were merged in commits only (274). If we were to include these 7,846 pull requests as merged, the percentage of merged pull requests grows from 45% to 55% for the entire dataset.

Peril X Not all activity is due to registered users: The impact of this peril is marginal: only 1.1% of committers in the dataset are not registered users.

Peril XI Only the user's public activity is visible: This is an overarching peril inherent in GitHub datasets due to the fact that they contain data from public repositories.

Peril XII GitHub's API does not expose all data: Some of the projects in the dataset have moved and are no longer active. The root repository of homebrew was renamed from *mxcl/homebrew* to *homebrew/homebrew*, although all the information was moved and is still available. Both *mangos/MaNGOS* and *TTimo/doom3.gpl* are dead. In the case of *mangos/MaNGOS*, the main trunk of the repository has been scrubbed of source code and moved to *cmangos/mangos-classic*, but the pull requests and issues of the old project were not moved to the new one. The author of *TTimo/doom3.gpl* keeps the repository for archival purposes; its development appears to have moved to *dhewm/dhewm3*. Other repositories have moved: *facebook/php-sdk* moved to *facebook/facebook-php-sdk* and no longer exists (it was not renamed). The *watchers* table of the dataset contains a field *created_ at* but this field corresponds to the date in which the watcher was discovered by GHTorrent, not the date the person became a watcher (this information is not exposed by GitHub's API, as described in *Peril XII GitHub's API does not expose all data*).

Peril XIII Github is continuously evolving: In the MSR dataset, watchers correspond to today's "starrings". This is because "watchers" are now a subscription mechanism, as explained in section 4.8.

5.2 The Papers

In light of the issues we have outlined above, we can illustrate the use of the perils for identifying potential threats to validity or offering alternative explanations. We use select papers published in the MSR Challenge of MSR'14 as examples and comment on some of the assumptions these papers made, contrasting them to the perils discussed earlier. We note that we are not making claims as to the validity of the results in the papers since we have not replicated the studies; we leave that for future studies that attempt to replicate or extend those studies.

In Sheoran et al (2014), we looked at watchers on GitHub and assessed if and when watchers become contributors and what types of contributions watchers make to the repositories they watch. One of the research questions involved investigating how long it takes for watchers to contribute to the repository, in any form; this was done through using the *created_ at* field of Watchers. As mentioned in the previous section, the *created_ at* field corresponds to the date GHTorrent recognized a user as a watcher, not the date the user became a watcher. If GHTorrent did not capture these watcher events as they occurred, the difference in timings can have an effect on the time the study concludes it takes for a watcher to contribute to a project.

Rahman & Roy (Rahman and Roy, 2014) analyzed GitHub pull requests. The study compared successful and unsuccessful pull requests against factors such as discussion items, pull request history, and selected project and developer characteristics. The goal of the comparative analysis was to identify factors that play a role in the success or failure of pull requests. The analysis considered merged pull requests as successful, while marked-as-non-merged as unsuccessful. As described above (*Peril IX Many merged pull requests appear as non-merged* applied to the MSR Data set), a significant number of pull requests are merged but not marked-as-merged (e.g., *django/django*, *Bukkit CraftBukkit* and *mxcl/homebrew*). This issue could have impacted the results in the paper; a different number of successful and unsuccessful pull requests can lead to different conclusions about the influence of the identified factors. For example, the peril could explain the outliers in Figures 5 and 6 and also why languages like Ruby (*mxcl/homebrew*) and Java (*Bukkit/CraftBukkit*) have a low ratio of marked-as-non-merged pull requests.

Padhye et al. (Padhye et al, 2014) also analyzed GitHub pull requests operating under the same assumption that not-marked-as merged pull requests were non-merged. The study distinguished between core, external and mutant commits based on whether they were merged in the base repository. Respectively, the study labeled committers according to their commits to identify communities and characterize them. As we noted above, the actual proportion of merged pull requests in the dataset changes if we count pull requests that are not marked-as-merged as merged, going up from 45% to at least 55%. This fact could have an effect on the set of commits that are marked as mutant in the study and potentially also reduce the number of committers labeled as mutant.

Matragkas et al. (Matragkas et al, 2014) analyzed user activity in projects to cluster users into roles, investigating the structure of the ecosystem of open source communities on GitHub. In the study each repository is considered and referred to as a "project", regardless of whether it is a base repository or a fork of one (see Table 1 in Matragkas et al (2014)). The rationale behind this choice is that it is hard to determine if work done in a fork is collaboration with other repositories or independent work that will not be contributed to other repositories; hence it is safer to consider them as separate[12]. Some forks will indeed not contribute back to the base repository, but it is difficult to determine if they will not. *Peril I A repository is not necessarily a project* could influence the results of the study, since considering some or all forks as part of a larger project would likely create larger clusters. Furthermore, the analysis counted the number of issues and issue comments per user. Under *Peril IV Many active projects do not use GitHub exclusively*, the size of the clusters may be underestimated if projects are using an issue tracker that is external to GitHub.

[12] The authors clarified this view in private communication.

The examples above demonstrate the potential threats to validity that the perils pose. This does not mean that the studies we critiqued (or others that use the same data) are flawed. Rather, it highlights that there are issues that need consideration when processing the data and drawing conclusions from it, and that need to be acknowledged in the discussion of a study's threats to validity.

6 Comparing perils between SourceForge and GitHub

Publicly available repositories are attractive data sources for researchers, but not without perils. There have been previous studies taking a critical look into the quantity and quality of data on public sources, with the most notable example being SourceForge. Over a decade ago, James Howison and Kevin Crowston (Howison and Crowston, 2004) identified perils and pitfalls in mining data from projects hosted on SourceForge. These perils related to three areas: data collection, interpretation and analysis, and research design. In Table 5 we highlight the similarity of our conclusions.

Table 5 Comparison between perils identified in (Howison and Crowston, 2004) and our study.

SourceForge perilous areas	**GitHub perils**
Data Collection	
- Spidering - Parsing - Summarizing - Testing	Data collection and summarizing relates to the GHTorrent dataset, explained in (Gousios, 2013) and (Gousios and Zaidman, 2014b)
Interpretation	
- Cleaning dirty data	P IV, P V, P VI, P X
- Skewed data	P I, P II, P III, P VII, P IX
Research design	
	P VIII, P XI, P XII, P XIII

In this paper we have not concerned ourselves with data collection perils because we used an already existing dataset, GHTorrent. In contrast, Howison & Crowston constructed their own dataset and, therefore, came across challenges and tradeoffs on how to mine the data in the first place, before analyzing it. The same applies to other studies that have used SourceForge data (e.g Weiss (2005)). The assumptions and heuristics in GHTorrent are described and assessed in previous work (Gousios, 2013; Gousios and Zaidman, 2014b).

Regarding the interpretation and analysis of data mined from SourceForge, Howison & Crowston recognized two challenging sub-areas: cleaning dirty data, and skewed data. In cleaning dirty data, similar to our conclusions, the authors observed that manual checking is essential since there is a lot of anonymous data (similar to our *Peril X: Not all activity is due to registered users*) while for many projects SourceForge may be the "repository of record" but not the "repository of use" (similar to our *Peril VI: Many active projects do not use GitHub exclusively*). Our *Perils IV* and *V* (*Many projects are not software development* and *Most projects are personal*) also relate to cleaning dirty data, since repositories would require manual inspection to categorize them properly.

Another peril in interpreting data from SourceForge (Howison and Crowston, 2004; Weiss, 2005) is how skewed the data is, which has also been our observation after reviewing GitHub data. Researchers need to be conscious of the data skewness and the fact that they will need to use screening variables to get data that is relevant to and representative of the properties they want to study, but also that the use of screening variables will significantly reduce the number of repositories and projects studied. We made the same observation relative to five of our perils too, noted in Table 5.

Finally, Howison & Crowston suggested caution to researchers designing studies using SourceForge data; the website provides a few easy-to-compute variables calculated for projects (the authors call them "ready-made"), but researchers use them to draw conclusions for complex theoretical constructs. This is a validity threat in itself, complicated by the fact that different literature areas may use the same variables as proxies for different concepts. The same caution applies in the case of GitHub data too. We also concluded that the simplicity of metrics may hide dangers; the number of commits in a pull request, for example, can be a simple metric to calculate but, given that GitHub does not report the intermediate commits that led to a merged pull request, could be a problematic proxy for the effort that was put in a successful merge. Perils XI, XII, and XIII also need to be taken into account in any research design so that conclusions do not include misinterpretations.

Surprisingly, the perils identified over ten years ago about the interpretation of data mined from public repositories and the research design of studies that build on that data are equally relevant today. Even though the data sources may have changed, researchers still have to be careful in how they extract data, how they analyze and interpret it, and how they make conclusions about software development.

7 Threats to Validity

Our study has several limitations and threats to validity. The exploratory survey had a relatively low number of participants from a biased and self-selected population. While it motivated us to investigate the perils in more detail, we can draw no further conclusions from it. Our manual exploration of 434 projects illustrates the variety of uses of GitHub, but we do not generalize our results to other projects.

This study was based on analysis of the GHTorrent dataset, and therefore, the reliability of our work depends partly on the reliability of the GHTorrent dataset. GHTorrent is a best-effort approach to collect data from the GitHub API and previous work (Gousios, 2013) analyzed the reasons why

GHTorrent cannot be a full replica of GitHub. The accuracy of the heuristics to detect pull requests merged outside GitHub is detailed in (Gousios and Zaidman, 2014b).

We mitigated these threats by triangulating quantitative with qualitative data from surveys, interviews and manual inspections.

To ameliorate these threats, we provide a replication package for our study. The package contains the results of our manual analysis as well as other data and scripts used in this work. The GHTorrent data is publicly available[13].

The replication package of this paper is available at `http://turingmachine.org/gitMiningPerils2014`.

8 Discussion & Conclusions

The story told by mined data is not always the whole story. This has been a finding in studies that assess the quality and completeness of data mined from project archives, but also in rare cases where the mined data is compared to qualitative evidence (Aranda and Venolia, 2009).

In this empirical study, we set out to critically look at the publicly available data coming from GitHub and assess whether it is suitable as a data source for software engineering studies. The data can be readily used to report on several project properties. If a researcher seeks to see trends of programming language use, type of tools built, number and size of contributions, and so on, the publicly available data can give solid information about the descriptive characteristics of the GitHub environment. However, using GitHub to synthesize information to draw conclusions about more abstract constructs needs some consideration. We presented evidence of how assumptions about repository activity and contents, as well as development and collaboration practices, can be challenged. We recommend that researchers interested in performing studies using GitHub data first assess its fit and then target the data that can really provide information towards answering their research questions.

Some potential perils manifest relative to the repository activity. One of the biggest threats to validity to any study that uses GitHub data indiscriminately is the bias towards personal use. While many repositories are being actively developed on GitHub, most of them are simply personal, inactive repositories. Therefore, one of the most important questions to consider when using GitHub data is what type of repository one's study needs and to then sample suitable repositories accordingly.

While we believe there to be a need for research on the identification and automatic classification of GitHub projects according to their purpose, we suggest a rule of thumb. In our own experience, the best way to identify active software development projects is to consider projects that, during a recent time period, had a good balance of number of commits and pull requests, and

[13] `http://ghtorrent.org/downloads.html`

have a number of committers and authors larger than 2. The number of issues can also be used as an indicator, but not all active projects use GitHub's issue tracker, such as several Mozilla projects[14]. Outliers, especially those with a very large number of commits per committer, point towards automatic bots.

When looking at any specific project, researchers need to keep in mind that other repositories might exist in the project—some of them working towards a common goal and some possibly being independent versions that will never contribute back. Based on our work, we believe a simple way to determine whether a repository actively works with another might be to identify if commits have flown from one to the other in both directions, but this strategy requires further validation.

Other potential perils manifest relative to the users and their characteristics. User actions might be taking place elsewhere and recorded as activity on GitHub, and due to non-unification of email addresses, not all of a user's activity is necessarily attributed to them. Both facts can distort the image researchers form of user activity and, therefore, potentially influence their conclusions. It is important to look more closely at the users' characteristics in light of the presented perils before drawing inferences and/or be aware of the potential threats to validity.

Apart from an exciting data source, GitHub is also an evolving entity. Its range of features and the integration between them changes frequently, meaning that the API also changes. This cannot be considered a flaw or attributed to GitHub as such; it simply puts more responsibility on researchers to remain aware of changes and factor them in their analysis. The same applies to information that is part of GitHub's functionality, yet reported partially or not at all.

One last conclusion point is to advocate for complementing quantitative studies with qualitative data. By all evidence we have presented, there are shortcomings in the data that can pose a danger to the conclusions of any rigorous study. Especially given *Peril XII GitHub's API does not expose all data*, researchers may not even be able to have direct access to information that could inform their interpretation of project or user activity. Getting additional qualitative input regarding projects and users can give more confidence in the assumptions that researchers make. For example, surveys that solicit comments from participants could be a solid information source.

We showcased the potential impact of the perils in a familiar and appropriate setting: the MSR 2014 Mining Challenge. The take-away message is that perils in the data equals perils in assumptions equals perils in results. We provided examples from our own and others' studies in the hope that researchers that continue to mine GitHub will be cautious of the underlying assumptions and informed about potential validity threats.

GitHub is a remarkable resource. It continues to grow at an accelerated rate and its users are finding innovative ways to exploit it. Nevertheless, software

[14] `https://github.com/mozilla`

development is flourishing in the open within GitHub's infrastructure and will continue to be an attractive source to mine for research in software engineering.

Acknowledgements We would like to thank the authors of Padhye et al (2014) and Matragkas et al (2014) for their valuable feedback regarding the evaluation of the impact of these perils on their research. We would also like to thank Margaret-Anne Storey for her invaluable help in the development of this paper.

References

Aranda J, Venolia G (2009) The secret life of bugs: Going past the errors and omissions in software repositories. In: Proc. of the 31st Int. Conf. on Software Engineering, pp 298–308

Bacchelli A, Bird C (2013) Expectations, outcomes, and challenges of modern code review. In: Proc. Int. Conf. on Soft. Eng.p, ICSE '13, pp 712–721

Bachmann A, Bird C, Rahman F, Devanbu P, Bernstein A (2010) The missing links: bugs and bug-fix commits. In: Proc. of the 18th ACM SIGSOFT international symposium on Foundations of software engineering, pp 97–106

Baysal O, Gousios G (2014) The MSR'14 Mining Challenge. `http://2014.msrconf.org/challenge.php`

Begel A, Bosch J, Storey MA (2013) Social networking meets software development: Perspectives from github, msdn, stack exchange, and topcoder. Software, IEEE 30(1):52–66

Bird C, Bachmann A, Aune E, Duffy J, Bernstein A, et al (2009a) Fair and balanced?: bias in bug-fix datasets. In: Proc. of the the Symposium On The Foundations Of Software Engineering, pp 121–130

Bird C, Rigby PC, Barr ET, Hamilton DJ, German DM, Devanbu P (2009b) The promises and perils of mining git. In: Mining Software Repositories, (MSR'09), IEEE, pp 1–10

Bissyande TF, Lo D, Jiang L, Reveillere L, Klein J, Le Traon Y (2013) Got issues? who cares about it? a large scale investigation of issue trackers from github. In: Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on, IEEE, pp 188–197

Corbin J, Strauss A (2008) Basics of qualitative research: Techniques and procedures for developing grounded theory. Sage

Dabbish L, Stuart C, Tsay J, Herbsleb J (2012) Social coding in GitHub: transparency and collaboration in an open software repository. In: Proc. Conf. on Computer Supported Cooperative Work, pp 1277–1286

Finley K (2011) Github Has Surpassed Sourceforge and Google Code in Popularity. `http://readwrite.com/2011/06/02/github-has-passed-sourceforge`

Gousios G (2013) The GHTorrent dataset and tool suite. In: Proceedings of the 10th Conference on Mining Software Repositories, MSR '13, pp 233–236, URL `http://dl.acm.org/citation.cfm?id=2487085.2487132`

Gousios G, Spinellis D (2012) GHTorrent: GitHub's data from a firehose. In: MSR '12: Proc. of the 9th Working Conf. on Mining Software Repositories, pp 12–21

Gousios G, Zaidman A (2014a) A dataset for pull-based development research. In: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, pp 368–371

Gousios G, Zaidman A (2014b) A dataset for pull-based development research. In: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, pp 368–371

Gousios G, Pinzger M, Deursen Av (2014) An exploratory study of the pull-based software development model. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pp 345–355

Gousios G, Zaidman A, Storey MA, Deursen Av (2015) Work practices and challenges in pull-based development: The integratorâĂŹs perspective. In: Proceedings of the 37th International Conference on Software Engineering, ICSE 2015, to appear

Grigorik I (2012) The Github archive. `http://www.githubarchive.org/`

Howison J, Crowston K (2004) The perils and pitfalls of mining sourceforge. In: Proc. of the Int. Workshop on Mining Software Repositories, pp 7–11

Kalliamvakou E, Damian D, Singer L, German DM (2014a) The Code-Centric Collaboration Perspective: Evidence from GitHub. Tech. Rep. DCS-352-IR, University of Victoria

Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014b) The promises and perils of mining github. In: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, pp 92–101

Kochhar PS, Bissyandé TF, Lo D, Jiang L (2013) Adoption of software testing in open source projects–a preliminary study on 50,000 projects. In: Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on, IEEE, pp 353–356

Marlow J, Dabbish L, Herbsleb J (2013) Impression formation in online peer production: activity traces and personal profiles in github. In: Proc. Conf. Computer Supported Cooperative Work, pp 117–128

Matragkas N, Williams JR, Kolovos DS, Paige RF (2014) Analysing the 'biodiversity' of open source ecosystems: The github case. In: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, pp 356–359

McDonald N, Goggins S (2013) Performance and participation in open source software on github. In: CHI'13 Extended Abstracts on Human Factors in Computing Systems, ACM, pp 139–144

Neath K (2012) Notifications & stars. `https://github.com/blog/1204-notifications-stars`

Nguyen TH, Adams B, Hassan AE (2010) A case study of bias in bug-fix datasets. In: Reverse Engineering (WCRE), 2010 17th Working Conference on, IEEE, pp 259–268

Padhye R, Mani S, Sinha VS (2014) A Study of External Community Contribution to Open-source Projects on GitHub. In: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, pp 332–335

Pham R, Singer L, Liskin O, Figueira Filho F, Schneider K (2013) Creating a shared understanding of testing culture on a social coding site. In: Proc. Int. Conf. on Soft. Eng., ICSE '13, pp 112–121

Rahman F, Posnett D, Herraiz I, Devanbu P (2013) Sample size vs. bias in defect prediction. In: Proc. of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp 147–157

Rahman MM, Roy CK (2014) An Insight into the Pull Requests of GitHub. In: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, pp 364–367

Rainer A, Gale S (2005) Evaluating the quality and quantity of data on open source software projects. In: Proceedings of the First International Conference on Open Source Systems (OSS 2005), pp 29–36

Rigby PC, Bird C (2013) Convergent contemporary software peer review practices. In: Proc. of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pp 202–212

Rigby PC, German DM, Storey MA (2008) Open source software peer review practices: a case study of the Apache server. In: Proce. of the 30th Int. Conf. on Software engineering, ICSE '08, pp 541–550

Sheoran J, Blincoe K, Kalliamvakou E, Damian D, Ell J (2014) Understanding "watchers" on github. In: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, pp 336–339

Takhteyev Y, Hilts A (2010) Investigating the geography of open source software through github. `http://takhteyev.org/papers/Takhteyev-Hilts-2010.pdf`

Thung F, Bissyande T, Lo D, Jiang L (2013) Network Structure of Social Coding in GitHub. In: 17th European Conference on Software Maintenance and Reengineering (CSMR), pp 323–326, DOI 10.1109/CSMR.2013.41

Tsay J, Dabbish L, Herbsleb J (2014) Influence of social and technical factors for evaluating contribution in github. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pp 356–366

Tsay JT, Dabbish L, Herbsleb J (2012) Social media and success in open source projects. In: Proc. Computer Supported Cooperative Work Companion, pp 223–226

Wagstrom P, Jergensen C, Sarma A (2013) A network of rails: a graph dataset of ruby on rails and associated projects. In: Proc. of the 10th Int. Work. Conf. on Mining Software Repositories, pp 229–232

Weiss D (2005) Quantitative analysis of open source projects on sourceforge. In: Proc. of the First Int. Conf. on Open Source Systems (OSS 2005), pp 140–147