The Promises and Perils of Mining GitHub

Eirini Kalliamvakou
University of Victoria
ikaliam@uvic.ca

Georgios Gousios
Delft University of Technology
G.Gousios@tudelft.nl

Kelly Blincoe
University of Victoria
kblincoe@acm.org

Leif Singer
University of Victoria
lsinger@uvic.ca

Daniel M. German*
University of Victoria
dmg@uvic.ca

Daniela Damian
University of Victoria
danielad@cs.uvic.ca

ABSTRACT

With over 10 million `git` repositories, GitHub is becoming one of the most important source of software artifacts on the Internet. Researchers are starting to mine the information stored in GitHub's event logs, trying to understand how its users employ the site to collaborate on software. However, so far there have been no studies describing the quality and properties of the data available from GitHub. We document the results of an empirical study aimed at understanding the characteristics of the repositories in GitHub and how users take advantage of GitHub's main features—namely commits, pull requests, and issues. 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. We show, for example, that the majority of the projects are personal and inactive; that GitHub is also being used for free storage and as a Web hosting service; and that almost 40% of all pull requests do not appear as merged, even though they were. We provide a set of recommendations for software engineering researchers on how to approach the data in GitHub.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Management—*Software configuration management*

General Terms

Software Engineering

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. GitHub introduced a "fork & pull" model in which developers create their own copy of a repository and submit a pull request when they want the project maintainer to pull their changes into the main branch. In addition to code hosting, collaborative code review, and integrated issue tracking, GitHub has integrated social features. Users are able to subscribe to information by "watching" projects and "following" users, resulting in a feed of information on those projects and users of interest. Users also have profiles that can be populated with identifying information and contain their recent activity within the site.

With over 10.6 million repositories[1] hosted as of January 2014, GitHub is currently the largest code hosting site in the world. Its popularity, the integrated social features, and the availability of metadata through an accessible API have made GitHub very attractive for software engineering researchers. Existing research has been both qualitative [4, 7, 16, 17, 19] and quantitative [10, 24, 25, 26]. Qualitative studies have focused on how developers use GitHub's social features to form impressions and draw conclusions on other developers' and projects' activity to assess success, performance, and possible collaboration opportunities. Quantitative studies have aimed to systematically archive GitHub's publicly available data and use that to investigate development practices and network structure in the GitHub environment.

As part of our research on collaboration on GitHub [15], 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. Through 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 host personal projects, without any plans to collaborate on their work. This signalled that there might be unseen, significant 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 risks of misinterpretation in publicly mined data has also been noted with regard to SourceForge mined data [14]. Furthermore, Bird et al. [6] described the promises associated with exploiting the information stored in a decentralized version control system. We, therefore formulated the following research question to address with this study:

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

*Corresponding Author

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

We highlight biases that can be found in the data and areas of the data that are easily misunderstood, side-by-side with the promises that GitHub data offers. We use insight gained from conducting a survey with 240 GitHub users to identify potential perils to assess, and we provide evidence of those perils through quantitative analysis on the GHTorrent dataset as well as a manual inspection of 434 GitHub repositories. We provide recommendations to researchers on how to best use the data available from GitHub and outline some analysis risks to avoid.

2. BACKGROUND & RELATED WORK

Many of the projects hosted on GitHub are public, thus anyone with an Internet connection can view the activity within those projects. The available activity includes actions around issues, pull requests, and commits including comments and subscription information. The large amount of public data on GitHub makes it possible for researchers to easily mine the project data, and various tools and datasets have been created to assist researchers in that goal.

2.1 Background

Web-based code hosting services such as GitHub have previously been of interest to software engineering researchers. The abundance and public availability of data simplifies some of the issues of data collection and processing. However, practical difficulties are still present and can potentially alter the conclusions drawn from the data.

SourceForge is another code hosting site; it peaked in popularity prior to GitHub's wide-spread adoption [8]. Howison and Crowston [14] 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 hosted outside of SourceForge space. Similarly, Weiss [28] concluded that not all SourceForge data is to be considered perfect: names of categories often change in SourceForge, and projects are constantly initiated and go inactive. By comparing his data to that of FLOSSMole[2], Weiss highlighted that information about inactive and inaccessible projects was missing altogether. In the same vein, Rainer and Gale [21] conducted an in-depth analysis of SourceForge data quality. They noted that only 1% of SourceForge projects were actually active as indicated by their metrics. The authors suggested caution in using of 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 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 these 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. [5] found that this set of bugs is a biased sample of the entire population. Bachmann et al. [3] 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. [18] discovered that similar biases exist even in commercial projects that employ strict guidelines and processes. More recently, Rahman et al. [20] 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.

2.2 Related work

Others have previously studied GitHub. The introduction of social features in a code hosting site has drawn particular attention from researchers. Several qualitative studies have interviewed GitHub users to better understand how these social features are being used [4, 7, 16]. Findings indicate that GitHub users form impressions and draw conclusions about other developers' and projects' activities and potential. 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. [19] 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. [26] performed a quantitative study on 5,000 projects to understand how GitHub's social features impact project success. McDonald and Goggins [17] 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. [25] built social networks of developers involved with 100,000 GitHub projects to demonstrate the social structure of the GitHub ecosystem. Takhteyev et al. [24] looked at the geographic locations of GitHub developers by examining self-reported location information available within GitHub profiles. Gousios et al. [10] 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 ones are only a small minority.

Other research has focused on making the data available through the GitHub API more readily available. The GHTorrent [11] project provides a mirror of the GitHub API data. It obtains its data by monitoring and recording GitHub events as they occur and applying recursive dependency-based retrieval of the related resources. The dataset provided by the GHTorrent project can be queried offline. It provides a comprehensive dataset since its data collection began in 2012, and it is currently working to retrieve all available history for important projects. When run in standalone mode, GHTorrent can also retrieve the history of individual repositories. The GitHub archive [13] provides a dataset of the history of events in GitHub. It also obtains its data by monitoring the GitHub timeline. However, it only contains events since its data collection began in 2011.

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

Moreover, one can use tools such as Gitminer [27] to extract the history of events for specific repositories. Gitminer crawls the GitHub API for any desired project and produces a graph dataset.

The popularity of GitHub and the ease with which data can be obtained is likely to see even more research targeting GitHub projects and users. The recommendations provided by our work can help researchers avoid common pitfalls associated with GitHub data.

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 [15]. During this study we surveyed GitHub users and then conducted interviews to further explore our study findings. Survey participants were selected 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 development process. We sent our survey to 1,000 GitHub users and received 240 responses (24%response rate). We received several responses out of the ordinary regarding the purpose of using GitHub. For example, respondants noted they used GitHub for purposes other than code hosting or collaborative development. These cases motivated our further analysis of the GitHub repository contents and collaboration within GitHub, as discussed in sections 4.2 and 4.3.

We then performed both quantitative and qualitative analyses of GitHub data to identify and measure the extent and frequency of the perils. Our process was divided in the following parts:

1. ***Quantitive analysis of project metadata.*** We used the GHTorrent [11] 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. In some cases, we cloned the repositories to further study their logs and file contents.

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 list of 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 nine perils to be aware of when analyzing GitHub data. Table 1 summarizes the perils. For each peril, we provide quantitative evidence to support it and, where appropriate, qualitative evidence as well. This section describes each peril in detail and provides quantitative and/or qualitative evidence showing their severity. We also provide recommendations on how to avoid each peril.

Peril I: A repository is not necessarily a project.

Pull requests as a distributed development model in general, and as implemented by GitHub in particular, form a new method for collaborating on distributed software development. In the pull-based development model, the project's main repository is not writable by potential contributors. Instead, these 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 to inspect the changes and pull them to 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 that case, contributors need to update their local branches with new commits.

As a consequence of this popular development model, we can divide repositories into two types: base repositories (those that are not forks) and forked repositories. The activity in forked repositories is recorded independently from their associated base-repositories. When a commit is made and is pulled into another repository via a GitHub pull-request, this commit does not appear in the history of the recipient repository; it only appears in the repository where the commit orginated. Therefore, measuring the activity of a repository independently of its forked repositories will ignore the activity of all of them as part of a single project.

For example, the Ruby on Rails project[4] has had 8,327 forks (8,275 forks were made directly from its base repository, and the remaining are forks of forks). Of the 50k commits in Rails `git` logs, GitHub reports only 34k commits as having occurred in the Rails base repository (rails/rails), and the remaining 16k as originating in its forks, plus another 11k that are in its forks but have not been propagated to the base repository. For example, the commit `fd1c515d4dac...` was found in the `git` repository of Rails. However, according to GitHub it was created in *arunagw/rails* and not in *rails/rails*. GitHub records that this commit was merged into *rails/rails* via a pull-request with id *13502*.

In the remaining of this paper, to properly account for all the activity of a software development team, 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, and only .6M have been forked at least once; thus these base repositories represent 3.0M different projects. For the base repositories with at least one fork, their number of forks is highly skewed: 80% have one fork only and those with at most 3 forks account for 94%. 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*, a GitHub administered repository for users to test how forking works.

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

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

[4] `http://rubyonrails.org` with GitHub repo located at `https://github.com/rails/rails`.

Peril	Description
I	A repository is not necessarily a project.
II	Most projects have very few commits.
III	Most projects are inactive.
IV	A large portion of repositories are not for software development.
V	Two thirds of projects (71.6% of repositories) are personal.
VI	Only a fraction of projects use pull requests. And of those that use them, their use is very skewed.
VII	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.
VIII	Most pull requests appear as non-merged even if they are actually merged.
IX	Many active projects do not conduct all their software development in GitHub.

Table 1: Summary of the perils discovered in our study.

4.1 On the Activity of Projects

Activity in GitHub is mostly reflected in commits. 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 very few commits.

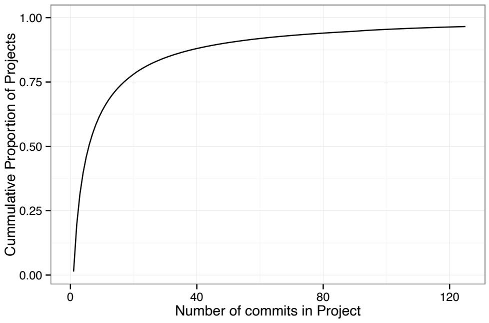

Figure 1: Cumulative ratio of projects with a given number of commits. Most projects have very few commits. The median number of commits per project is 6, and 90% of projects have less than 50 commits.

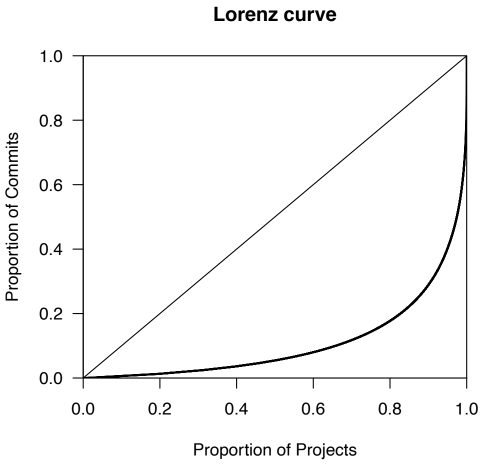

Figure 2: Lorenz curve showing that very few projects account for most of the commits.

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.

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 III: Most projects are inactive.

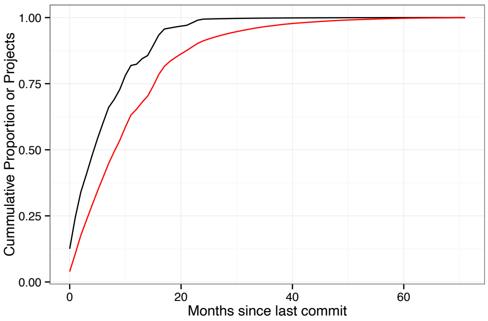

Figure 3: Cumulative ratio of active projects during the last n months since Jan 9, 2014. The red line is the proportion of projects created during the last n months. Approximately 46% of projects have been inactive in the last six months. Only 13% of projects were active in the last month, and 1/3 of them were created during that period.

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 of creation of its first repository in GitHub with the date of the project's last commit. This is shown in Figure 4. In this regard, the median number of days a project is active is 9.9 days. 32% of projects have only been active for one day, suggesting that they are being used either for testing or for archival purposes. Only 38% have been active for more

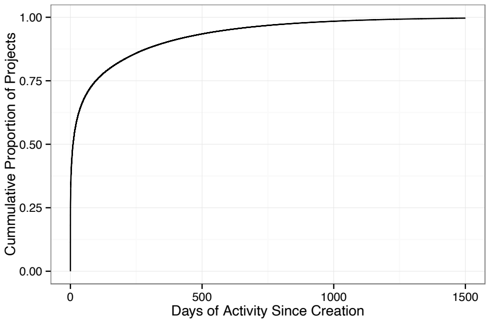

Figure 4: Cumulative ratio of projects that had activity the last n days since their creation. The median number of days is 9.9, with 25% of projects at 100 or more days; only 32% have activity less than one day after they have been created.

than one month. However, active projects continue to be active: 25% of projects have activity for at least 100 days.

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

4.2 On the Contents of Projects

Peril IV: A large portion of repositories are not for software development.

The answers to the survey indicated that GitHub is used for a number of purposes besides software development. Thirty-four of our 240 respondents (14%) said they use GitHub repositories for experimentation, hosting their websites, and for academic/class projects. About 10% of respondants use GitHub specifically for storage.

The purpose of a repository cannot be reliably and automatically identifed from the project metadata. We used the 434 randomly selected repositories to determine if GitHub repositories are used for software development or other purposes. We reviewed the description and files of each repository and assigned it an appropriate label to mark its contents, e.g "software library" or "class project". Next we aggregated the labels to 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 includes repositories that contain configuration files (including "."' files) or other documents and files for personal use, such as presentation slides, resumes and such. Repositories that gave a 404 error 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". The categories and the distribution of the 434 repositories are shown in Table 2.

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%)

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

In particular, Web has become an important use of GitHub. GitHub allows it users to host websites on its servers for free[5]. 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 selecting projects to analyze, one should not rely only on the types of files within their repositories to identify software development projects. Researchers should review the description and README file to ensure the project fits their research needs.

4.3 On the Users Involved with Projects

Peril V: Two thirds of projects (71.6% of repositories) are personal.

Our survey asked respondents if they used GitHub primarily for collaboration with others or personal use. 90 out of 240 respondents (38%) answered that they use GitHub primarily 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 and its committer. The committer is the person who has right access to a repository. In GitHub, only 2.9% of commits have an author who is not its 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 one committer, 87% have two or less, and 93% three or less. As expected, repositories have less committers than projects: 72% have one committer, 91% have 2 or less, and 95% 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, the number of committers should be considered.

4.4 On Pull Requests

Promise I: 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

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

code hosting sites improved this process significantly by integrating code reviews, discussions and issues, thus effectively lowering the entry barrier for casual contributions. Combined, 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.

Peril VI: Only a fraction of projects use pull requests. And of those that use them, their use is very skewed.

Across GitHub, the use of pull requests is not very widespread. Pull requests are only useful between developers, thus, nonexistent in personal projects (67% of projects, see Peril V). Of the 2.6 million projects that represent actual collaborative projects (at least 2 committers) only 268,853 (10%) used the pull request model at least once to incorporate commits; the remaining 2.4 M projects would have used GitHub in 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 exist projects, such as the Gaia phone application framework and the Homebrew package manager that received more than 5,000 pull requests in 2013 alone. In fact, a significant number of projects (∼1700) received more than 100 pull requests in 2013, which make a sample big enough to deliver statistically significant results for many research questions.

Peril Avoidance Strategy: When researching the code review process on GitHub, the number of pull requests must be considered when selecting appropriate projects.

4.4.1 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 base ("upstream" in `git` parlance) 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 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 could be inferred from the comments.

In case of an update, the contributor creates new commits in the forked repository and, after the changes are pushed to the branch to be merged, GitHub will automatically update the commits in the pull request. The code review can then be repeated on the refreshed commits. In our 434 project dataset, 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 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 that 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 in code review [23, 22, 2], which suggests that the peer review process may have more fundamental underpinnings yet to be explored. The GitHub data may then 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. For example, many pull requests (46% in our 434 project sample) receive no code comments and no discussion, while they are still merged. It is usually safe to expect that the developer that did the merge did inspect the pull request before merging it (unless it is project policy to accept any pull request without reviewing).

Peril VII: 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.

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. It is common in projects to 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 review process.

Peril Avoidance Strategy: To perform analysis on the full set of commits involved in a code review, researchers must not rely on the commits reported by GitHub.

Peril VIII: Most pull requests appear as non-merged even if they are actually merged.

When 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 hand selected commits from the pull request branch are merged to the main branch.
- By creating a textual patch between the pull request and main repository branches and applying to the master branch. This is also known as commit squashing.

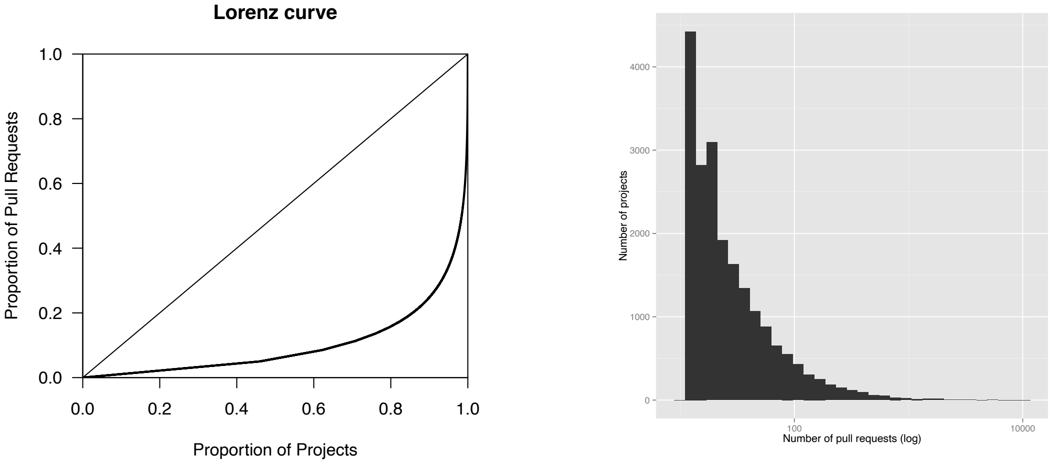

Figure 5: Lorenz curve for the number of pull requests per project (left) and the corresponding histogram (right). The top 1.6% of projects use 50% of the total pull requests. These plots only include projects with at least one pull request.

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 examples 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[6]. 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 its 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 [10]).

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 3 (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 above.

Across GitHub, 1,145,099 of 2,552,868 or 44% of the pull requests are reported as merged. On 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 [10], where we used a carefully selected sample of 297 projects that heavily relied on pull requests, 65% of the pull requests where merged with GitHub facilities, while the heuristics identified another 19% (H_1: 7%, H_2: 1%, H_3: 3%, H_4: 7%) as merged. In yet another dataset [12], where we include almost 1000 projects that use pull requests, 58% of the pull requests are 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 might lead to false positives (especially H_4). In other work [10], we manually inspected 350 pull requests that where not identified as merged and found that 65 of them where 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, while heuristics can improve merge detection, in some cases dramatically.

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

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

4.4.2 Pull requests as an issue resolution mechanism

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

Issues and pull requests are dual 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 the 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, which helps projects to track progress.

The fact that issues and pull requests are so tightly integrated opens a window of opportunity for very detailed studies of developer activity. For example, a researcher can track the resolution of an issue from the reporting phase, through the modifications of the source code, 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 characteristics of issues 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.

4.5 On the Use of Non-GitHub Infrastructure

Peril IX: Many active projects do not conduct all their software development in GitHub.

A difficult question to answer is if 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 in their process?

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 can not 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. 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 other 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, it is composed of 15 different repositories.

We identified many repositories which are mirrors. GitHub itself has setup mirrors of many popular projects[7], and these account for 91 GitHub repositories. Usually 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 that 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 1,851 projects (12,709 repositories) as mirrors of other repositories outside GitHub. 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` hosting repository). We have summarized these results in Table 3.

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

The development within a mirror in GitHub of an external repository implies that some members of a project are using GitHub for one of two purposes: (1) to develop 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) contributes commits regularly to the linux kernel (we have observed 123 commits in Linus Torvald's repository that originated here[8]). (2) to develop customizations of the original project for a different purposes, independent of the original development team; in this category we find multiple repositories to develop 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

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

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

Set	Used regular expresion	No. Projets	No. Repos
Mirror Of	_mirror of .*repo|git mirror of_	1,851	12,709
Subsets			
Located at Sourceforge	_sourceforge|sf\\.net_	117	511
Located at 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

Table 3: Repositories hosted on GitHub labelled 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.

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 issues in GitHub, and expects users to file issues at `bugzilla.mozilla.org`.

We conducted a small survey in which we asked respondents to tell us whether they used GitHub's or external tools for a set of tasks, such as opening and merging pull requests, tracking issues, or for communication. We sent an online questionnaire to 100 GitHub users via email. Of those, 27 responded (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 which explicitly state that they are mirrors in their description.

5. THREATS TO VALIDITY

Our study has several 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.

The reliability of this study depends on the reliability of the GHTorrent dataset. GHTorrent is a best-effort approach to collect data from the GitHubAPI. Previous work [9] has 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 [12].

To ameliorate these threats, we provide a replication package for our study. The GHTorrent data is publicly available[9]. The results of our manual analysis as well as other data and scripts used in this work are available separately in the replication package.

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

6. 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 [1]. 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.

Using GitHub to synthesize information to draw conclusions about more abstract constructs though 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.

Perhaps the biggest threat 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 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[10]. 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

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

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

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.

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 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.

7. REFERENCES

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

[2] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proc. Int. Conf. on Soft. Eng.p*, ICSE '13, pages 712–721, 2013.

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

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

[5] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, and et al. Fair and balanced?: bias in bug-fix datasets. In *Proc. of the the 7th European Software Engineering Conference and the ACM SIGSOFT Symposium On The Foundations Of Software Engineering*, pages 121–130, 2009.

[6] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining git. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 1–10. IEEE, 2009.

[7] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Proc. of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 1277–1286, 2012.

[8] K. Finley. Github has surpassed sourceforge and google code in popularity. `http://readwrite.com/2011/06/02/github-has-passed-sourceforge`, 2011.

[9] G. Gousios. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Conference on Mining Software Repositories*, MSR '13, pages 233–236, 2013.

[10] G. Gousios, M. Pinzger, and A. van Deursen. An exploration of the pull-based software development model. In *ICSE '14: Proc. of the 36th Int. Conf. on Software Engineering*, June 2014. To appear.

[11] G. Gousios and D. Spinellis. GHTorrent: GitHub's data from a firehose. In *MSR '12: Proc. of the 9th Working Conf. on Mining Software Repositories*, pages 12–21, jun 2012.

[12] G. Gousios and A. Zaidman. A dataset for pull request research. In *Submitted to MSR '14 – data track*.

[13] I. Grigorik. The github archive. `http://www.githubarchive.org/`, 2012.

[14] J. Howison and K. Crowston. The perils and pitfalls of mining sourceforge. In *Proc. of the Int. Workshop on Mining Software Repositories (MSR 2004)*, pages 7–11, 2004.

[15] E. Kalliamvakou, D. Damian, L. Singer, and D. M. German. The code-centric collaboration perspective: Evidence from github. Technical Report DCS-352-IR, University of Victoria, February 2014.

[16] J. Marlow, L. Dabbish, and J. Herbsleb. Impression formation in online peer production: activity traces and personal profiles in github. In *Proc. Computer Supported Cooperative Work*, pages 117–128, 2013.

[17] N. McDonald and S. Goggins. Performance and participation in open source software on github. In *CHI'13 Extended Abstracts on Human Factors in Computing Systems*, pages 139–144. ACM, 2013.

[18] T. H. Nguyen, B. Adams, and A. E. Hassan. A case study of bias in bug-fix datasets. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 259–268. IEEE, 2010.

[19] R. Pham, L. Singer, O. Liskin, F. Figueira Filho, and K. Schneider. Creating a shared understanding of testing culture on a social coding site. In *Proc. Int. Conf. on Soft. Eng.*, ICSE '13, pages 112–121, 2013.

[20] F. Rahman, D. Posnett, I. Herraiz, and P. Devanbu. Sample size vs. bias in defect prediction. In *Proc. of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 147–157, 2013.

[21] A. Rainer and S. Gale. 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)*, pages 29–36, 2005.

[22] P. C. Rigby and C. Bird. Convergent contemporary software peer review practices. In *Proc. of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 202–212, 2013.

[23] P. C. Rigby, D. M. German, and M.-A. Storey. 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, pages 541–550, 2008.

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

[25] F. Thung, T. Bissyande, D. Lo, and L. Jiang. Network structure of social coding in github. In *17th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 323–326, 2013.

[26] J. T. Tsay, L. Dabbish, and J. Herbsleb. Social media and success in open source projects. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work Companion*, pages 223–226, 2012.

[27] P. Wagstrom, C. Jergensen, and A. Sarma. 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*, pages 229–232, 2013.

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