Knowledge Transfer in Modern Code Review

Maria Caulo[1], Bin Lin[2], Gabriele Bavota[2], Giuseppe Scanniello[1], Michele Lanza[2]

1: University of Basilicata, Italy, 2: REVEAL @ Software Institute - Università della Svizzera Italiana, Switzerland
Email:{maria.caulo,giuseppe.scanniello}@unibas.it,{bin.lin,gabriele.bavota,michele.lanza}@usi.ch

ABSTRACT

Knowledge transfer is one of the main goals of modern code review, as shown by several studies that surveyed and interviewed developers. While knowledge transfer is a clear expectation of the code review process, there are no analytical studies using data mined from software repositories to assess the effectiveness of code review in "training" developers and improve their skills over time. We present a mining-based study investigating how and whether the code review process helps developers to improve their contributions to open source projects over time. We analyze 32,062 peer-reviewed pull requests (PRs) made across 4,981 GitHub repositories by 728 developers who created their GitHub account in 2015. We assume that PRs performed in the past by a developer D that have been subject to a code review process have "transferred knowledge" to D. Then, we verify if over time (*i.e.,* when more and more reviewed PRs are made by D), the *quality* of the contributions made by D to open source projects increases (as assessed by proxies we defined, such as the acceptance of PRs, or the polarity of the sentiment in the review comments left for the submitted PRs). With the above measures, we were unable to capture the positive impact played by the code review process on the quality of developers' contributions. This might be due to several factors, including the choices we made in our experimental design.Additional investigations are needed to confirm or contradict such a *negative result.*

CCS CONCEPTS

• **Software and its engineering → Collaboration in software development**; **Software libraries and repositories**; • **Information systems →** *Sentiment analysis.*

KEYWORDS

knowledge transfer, code review, mining software repositories

ACM Reference Format:
Maria Caulo[1], Bin Lin[2], Gabriele Bavota[2], Giuseppe Scanniello[1], Michele Lanza[2]. 2020. Knowledge Transfer in Modern Code Review. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3387904.3389270

1 INTRODUCTION

Code review is the process by which peer developers inspect the code written by a teammate to assess its quality, to recommend changes and, finally, to approve it for merging [3]. Previous works have investigated code review from several perspectives. Some authors studied the factors influencing the likelihood of getting a patch accepted as the results of the code review process [5, 41], while others studied the reviewing habits of developers in specific contexts [34]. Several works focused on the benefits, motivations, and expectations of the review process. Most of these studies are qualitative in nature [2, 6, 33], and were conducted by surveying/interviewing developers or by inspecting their conversations in mailing lists or issue trackers of open source projects. Only a few researchers analyzed data from a quantitative perspective, mostly to assess the impact of code review on code quality (*e.g.,* the relationship between code review and post-release defects) [4, 20, 24, 25].

The work conducted at Microsoft by Bacchelli and Bird [2] provided qualitative evidence of the central role played by code review in knowledge transfer among developers. However, no quantitative, mining-based study has tried to investigate this phenomenon, and in particular to answer the following high-level research question (RQ): *Does code review enable knowledge transfer among developers?.*

Answering this RQ, by mining software repositories, is far from trivial since: *(i)* quantitatively measuring knowledge transfer is challenging and an open research problem by itself and *(ii)* many confounding factors come into play when collecting developer-related data from online repositories. We quantitatively answer the above research question by making the following assumptions:

- *The number of* ***reviewed*** *pull requests (PRs) a developer made in the past across all repositories she contributed to is a proxy of the transferred knowledge she benefited of.* Given a developer D, we assume that the higher the number of closed PRs (*i.e.,* accepted and rejected ones) that were subject to review (*i.e.,* received comments from peer developers) D performed, the higher the knowledge transfer D benefited of.
- *We can measure the actual benefits of the knowledge transfer experienced through the code review process by a developer, by observing if, with the increase of the received knowledge transfer, the quality of her contributions to open source projects increases as well.* Given the various types of projects involved, it is necessary to adopt contribution quality measures which are independent from project languages and domains. We assume that how code reviewers respond to developers' PRs can reflect the quality of the submitted contribution.We use as proxies for the quality of the contributions provided by D: *(i)* the percentage of D's PRs that are accepted (expected to increase over time); *(ii)* the time required to review the changes D contributes (expected to decrease); *(iii)* the amount of recommendations provided by the

reviewers to improve the code D contributes in PRs (expected to decrease); and *(iv)* through sentiment analysis, the polarity of the sentiment in the discussion of the PRs D submits (expected to be more positive).

Based on these assumptions, we analyzed the contribution history of 728 developers across 4,981 repositories hosted on GitHub. We studied whether the number of reviewed PRs opened in the past by a developer impacts the quality of her contributions over time.

We grouped developers into different sets based on the amount of knowledge transfer they benefited of (low, medium-low, medium-high, high), as assessed by the number of reviewed PRs they performed in the past. Any result achieved with such an experimental design may be due to a simple increase of the developer's experience over time rather than to the knowledge transfer that took place over the reviewed PRs. To control for this, we replicated our analysis by grouping the developers based on the number of commits rather than the number of reviewed PRs they performed in the past (into the four groups listed above). Using our experimental design with the measures mentioned above, we were not able to capture the positive impact played by the code review process on the quality of developers' contributions. Such a negative result might be due to several factors, including the choices we made in our experimental design (see Section 3). For this reason, additional studies are needed to corroborate or contradict our findings.

2 RELATED WORK

Recent works studying PR-based software development [13, 21, 31, 32, 35–37, 39] have focused on the motivations of acceptance or rejection of changes proposed in the form of PRs after the code review process, identifying various influencing factors, such as:

- **Programming Language**: proposed changes in Java are the least easily accepted, whereas for C, Typescript, Scala and Go the opposite happens [32], [36];
- **Size and Complexity of the PR**: the greater the size and complexity of the PR to be reviewed (*e.g.,* the number of the commits, or the committed files) the lower the likelihood of acceptance [39], [37], [35], [31], [21];
- **Addition and Change of files**: PRs which propose to add files have a 8% lower chance of acceptance [36]; the same applies for PRs which contain many changed files [31];
- **Excessive forking**: PR acceptance decreases when many forks are present [32];
- **Tests**: contributions including test code are more likely to be merged [39], [13];
- **Developer's type**: if the PR was made by a member of the core team, it has more chances to be accepted as compared to a PR made by an external. The existence of a social connection between the requester, the project and the reviewer, positively influences merge decisions [36], [39], [21];
- **Experience in making PRs**: the higher the percentage of previously merged PRs by a developer, the higher the chances of acceptance [13]. Developers with 20 to 50 months of experience are the most productive in submitting and being accepted their PRs [32]. When a PR is the first made by a developer, the chance of a merge considerably decreases [39], [37], [36], [21];
- **Number of comments**: the more comments have been made in the PR discussion, the lower the chance of acceptance [39], [35].

Bosu *et al.* [10] investigated which factors lead to qualitatively high code reviews. To discern if a code review feedback is useful or not, the authors built and verified a classification model, and executed it on 1.5 million review comments from 5 Microsoft projects, finding several factors that affect the usefulness of reviews feedback: *(i)* the working period of the reviewer in the company: in the first year she tends to provide more useful comments than afterward; *(ii)* reviewers from different teams gave slightly more useful comments than reviewers from the same team; *(iii)* the density of useful comments increases over time; *(iv)* source code files had the highest density of useful comments than other types of files; and *(v)* the higher the size of the change (*i.e.,* the number of files involved) that the author would bring to a project, the lower the usefulness of the review comments to such an author, confirming in some sense the results by Weißgerber *et al.* [41]. Weißgerber *et al.* studied the email archives of two open source projects to find which factors affect the acceptance of patches. They found that small patches (at most 4 lines changed) have higher chances to get accepted, but the size of a patch does not significantly influence acceptance time.

Baysal *et al.* [5] investigated which factors affect the likelihood of a code change to be accepted after code review. They extracted both "ordinary" factors (code quality-related) and non-technical ones, such as organizational (company-related) and personal (developers-related) features, finding that nontechnical factors significantly impact the code review outcome.

Company and developers-related factors of reviews practices (in open-source projects) have been qualitatively studied also by Rigby *et al.* [33, 34], who compared, by means of emails archives and version control repositories, the two techniques used by developers of Apache server project: review-then-commit and commit-then-review [33]. Apache reviews resulted to be early and frequent, related to small and completed patches (in line with Weißgerber *et al.* [41]), and conducted by a small number of developers. Rigby *et al.* [34] also investigated *(i)* the mechanisms and behaviours that developers use to find (or ignore) code changes they are competent to review and *(ii)* how developers interact with one another during the review process.

Research has also been conducted to study how software quality is impacted by code reviews, and how they allow to identify defects. Kemerer and Paulk [20] studied the review rate to adopt to have effective reviews when removing defects or influencing the software quality. The authors studied two datasets from a personal software process (PSP) approach with regression and mixed models. The PSP review rate turned out to be significant for the effectiveness of bug-fixing tasks. Mäntylä *et al.* [29] classified the issues found by both students and professional developers during code review. They found that 75% of issues concerned "evolvability" issues (*e.g.,* limited readability/maintainability of code). Beller *et al.* [6] confirmed this finding by classifying changes brought by the reviewed code of two open-source software projects. They found a 3:1 ratio between maintainability-related and functional defects. They also found that bug-fixing tasks need fewer changes than others, and the person who conducts the review does not impact the number of required changes. Czerwonka *et al.* [15] observed that code reviews often do not identify functionality problems. The authors found that code reviews performed by unskilled developers are not effective, highlighting the importance of social aspects in code review.

McIntosh *et al.* quantitatively studied the relationship between software quality and *(i)* the amount of changes that have been code reviewed, and, *(ii)* code review participation, *i.e.,* the degree of reviewer involvement in the code review process [24]. The authors studied three projects and found that both aspects are linked to software quality: poorly reviewed code leads to components with up to two post-release defects; low participation up to five. Bavota and Russo [4] studied the impact of code review on the quality of the committed code. They found that unreviewed commits have twice more chances of introducing bugs as compared to reviewed commits. Also, code committed after a review is more readable than unreviewed code.

Morales *et al.* [26] studied the effect of code review practices on software design quality. They considered the occurrences of 7 design and implementation anti-patterns and found that the lower the review coverage the higher the likelihood to observe those anti-patterns in code. Bernart *et al.* [7, 8] highlighted that continuous code review practices in agile development produce high benefits to a project, such as *(i)* the reduction of the effort in software engineering practices, *(ii)* the support of collective ownership; and *(iii)* the improvements in the general understandability of the code.

Recent research work also focused on the content of conversations deriving from the code review activity, the topic of the discussions, and how developers emotionally felt [16, 22, 30]. Li *et al.* [22] classified review comments according to a custom taxonomy of topics, finding that *(i)* PRs submitted by inexperienced contributors are likely to have potential problems even if they passed the tests; and *(ii)* external contributors tend to not follow project conventions in their early contributions. Destefanis *et al.* [16] analyzed GitHub issues commenters (*i.e.,* those users who only post comments without posting any issues nor proposing changes to repositories) from the effectiveness perspective. The authors found that commenters are less polite and positive, and express a lower level of emotions in their comments than other types of users. Ortu *et al.* [30] found that GitHub issues with a high level of Anger, Sadness, Arousal and Dominance are less likely to be merged, while high values of Valence and Joy tend to make issues merged.

Bacchelli and Bird [2] studied the tool-based code review practices adopted at Microsoft, reporting that even if finding defects remains the main motivation for reviews, they provide additional benefits, such as knowledge transfer, increased team awareness, and creation of alternative solutions to problems.

2.1 Taking Stock

The relevance of code reviews has been investigated from different perspectives. The effect of code reviews on knowledge transfer has been only marginally studied, let alone from a quantitative perspective, which is the goal of this paper: We used the number of past reviewed PRs submitted by a developer as a proxy for the amount of knowledge transfer she has been subject to. Then, we assess whether with the increase in received knowledge transfer, the quality of submitted code contributions improves over time. From this perspective, the most similar work is the recent one by Chen *et al.* [13], in which the authors found that the highest the percentage of previously merged PRs by a developer, the higher the chances of acceptance of new PRs.

Differently from Chen *et al.* [13], we consider past submitted PRs (both accepted and rejected) that have been actually reviewed (*i.e.,* received at least one comment from peer developers), to get a "reliable" proxy of the amount of knowledge transfer of a developer in the past. Also, besides analyzing the impact of the received knowledge transfer on the likelihood of acceptance for future submitted PRs, we consider many other proxies to assess the quality of the contributions submitted by a developer.

3 STUDY DESIGN

3.1 Hypothesis

Software development is a knowledge-intensive activity [9]. Qualitative research provided evidence that code review plays a pivotal role in knowledge transfer among developers [2]. However, no quantitative evidence exists in support of this claim. In this study, we mine software repositories to quantitatively assess the knowledge transfer happening thanks to code review.

There is no well-established metric to assess the "quantity of knowledge" involved in a given process. Knowledge can be classified as either *explicit* (which *"can be spoken and codified in words, figures or symbols"*) or *tacit* (which *"is embedded in individuals' minds and is hard to express and communicate to others"*) [1]. We focus on the tacit knowledge acquired by developers over time, which cannot be easily seen and quantified. More specifically, we investigate whether the experience gained by receiving feedback during code review improves the quality of developers' future contributions to open source projects. Intuitively, one might expect that developers gradually gain knowledge by receiving feedback from their peers, thus improving their skills over time. Therefore, we formulated and studied the following hypothesis:

> **H.** *The quality of developers' contributions to software projects will increase with the experience gained from their past reviewed PRs.*

3.2 Study Context

The *study context* consists of 728 developers, 4,981 software repositories they contributed to, and 77,456 closed PRs (among which 32,062 PRs are peer-reviewed).

3.2.1 *Developers selection.*

To run our study, we collected information about GitHub users (from here onward referred to also as developers), who created their account in 2015. This was done to collect at least four years of contribution history for each developer. Since data was collected in September 2019, we can observe ~4 years of contributions even for users who created their GitHub account in December 2015. A four-year time window is long enough to observe enough PRs submitted by developers and, consequently, to study the knowledge transfer over time.

We used the *GitHub Search API*[1] to retrieve the developers who joined GitHub on the first day of each month in 2015. Since the *GitHub Search API* only provides up to 1,000 results for search, we collected a total of 12,000 developers who created their account

[1]https://developer.github.com/v3/search/

in 2015 (*i.e.,* 1,000 per month). As the next step, we collected all the PRs submitted by these 12,000 developers across all GitHub repositories they contributed to.

Since the *GitHub Search API* cannot return over 1,000 PRs for a single developer, to ensure the data completeness, we excluded nine developers who submitted over 1,000 PRs in the studied time window. This reduced the number of developers to 11,991.

We removed from our dataset developers who submitted too few PRs. This was needed since we want to analyze how the quality of developers' contributions to open source projects changes over time. Having only one or two PRs submitted by a developer would not allow to perform such an analysis. For this reason, we excluded from our study all developers who submitted less than 30 PRs in the considered time period (*i.e.,* 2014-2019). This further filter removed 11,173 developers, leaving 818 developers in total.

3.2.2 Pull requests collection and filtering. We collected all the *"closed"* PRs submitted by the 818 subject developers from the day they joined GitHub until the end of September 2019, when we collected the data. This led to a total of 77,456 PRs spanning 9,845 repositories. We only focused on closed PRs to be sure that the PRs underwent a code review process and, thus, were either accepted or rejected instead of still pending. For each PR, we collected the following information:

(1) *Creation date:* the date in which the PR was submitted.
(2) *Acceptance:* whether the closed PR was accepted.
(3) *Closing date:* the date in which the PR was closed.
(4) *Source code comments:* the comments left by the reviewers that are explicitly linked to parts of the code submitted for review. Comments left by the PR author are excluded.
(5) *General comments:* all the comments left in the PR discussion by all the developers other than the PR author, excluding *source code comments.* These comments are generally used to ask for clarifications or to explain why a PR should be accepted/rejected. Source code comments, instead, reports explicit action items for the PR author to improve the submitted code. We separate the *source code comments* and the *general comments*, as there might be different levels of technical details in these two categories.
(6) *Author:* the author of the PR.
(7) *Contributors:* all the developers who have been involved in the discussion and handling of the PR.

Since we plan to use the comments related to each PR as one of the variables for our study, *i.e.,* to assess the amount of feedback received by developers as well as to check whether a PR was actually subject to code review (meaning, it received at least one comment), we removed general comments posted by bots (this problem does not occur for source code comments). We discriminated whether a comment was left by a bot following the steps below:

(1) We calculated how many general comments each commenter (*i.e.,* entity who posted at least one comment in the considered PRs) left in the PRs and sorted them in descending order. As a result, around 60% of the comments were left by the top-500 commenters, with a long tail of commenters only posting a handful of comments in their history.
(2) For these top-500 commenters, we manually checked their usernames and profile images. If the username contained "bot," or the profile image represented a robot, we then further inspected whether their comments followed a predefined structure, *e.g.,* "Automated fastforward with [GitMate.io] (https://gitmate.io) was successful!", by *gitmate-bot.* If this was the case, we considered the commenter as a bot.
(3) For the rest of the commenters, we manually checked the GitHub profiles of those whose username contained "bot".

This process led to the disclosure of 147 bot commenters. The manual identification of the bots was done by the first author, and the final output (*i.e.,* the 147 removed bots) is available in our replication package [12].

After this cleaning process, we further excluded 90 developers from our study since they authored less than 30 closed PRs (including those which did not receive comments). This led to the final number of 728 developers considered in our study, who authored a total of 77,456 PRs (among which 32,062 PRs received comments).

3.2.3 Project collection. We cloned all the projects[2] in which the selected developers submitted at least one PR, for a total of 4,981 repositories. To provide a better overview of the collected projects, our replication package[12] also includes basic information (*e.g.,* programming languages, project size) of these repositories.

3.3 Measures

To verify our hypothesis, we use proxies to measure the knowledge transfer experienced by developers through their past reviewed PRs and to assess the quality of developers' contribution over time.

3.3.1 Knowledge measures. We use the number of **reviewed** PRs a developer contributed (authored) in the past (*i.e.,* before the current PR) as a proxy of the amount of knowledge transferred to her thanks to the code review process. That is, we assume that the more closed and peer-reviewed PRs a developer has, the more knowledge the developer gained. In our study, we consider that peer-reviewed PRs are those which received at least one comment by non-bot users. The rationale behind this choice is that if no comments are given by other developers, we assume that the PR was not subject of a formal review process and, thus, it is not interesting for our goals, since no transfer knowledge can happen in that PR. We compute this number for each developer before each of their peer-reviewed PR. We use this variable to split developers into different groups based on the knowledge transfer they experienced (*i.e.,* low, medium-low, medium-high, and high), and compare the quality of the submitted contributions (as assessed by the proxies described in the following section) among the different groups. This means that the same developer can belong, in different time periods, to different groups (*i.e.,* she starts in the low transfer knowledge group, she then moves to medium-low, etc.). The exact process used for data analysis is detailed later on.

To verify whether the quality of the submitted contributions is actually influenced by the knowledge transfer during code review or if it is just a result of the increasing developer's experience over time, we also collected the number of commits performed in the past by each developer before submitting each PR. The commits are

[2]This was done since we also used in our analysis the number of commits performed by the studied developers over time. While this information can be collected through the GitHub APIs as well, cloning the repositories simplified data collection.

extracted from all repositories in which the developers submitted at least one PR. As done for the past PRs, we use past commits to split developers into groups and contrast the quality of their contributions over time.

This allows us to see whether potential differences in contribution quality among the groups can be attributed to the code review process put into place in PR (*i.e.,* these differences are visible when splitting developers based on past reviewed PRs, but not when splitting them based on past commits) or if they are mainly due to changes in the experience over time (*i.e.,* the differences can be observed both when splitting by past reviewed PRs as well as by past commits). When retrieving past commits for developers, there are two issues worth noting: 1) The developer's username on GitHub (as extracted using the GitHub API) might be different from the author name in the Git commit history (as extracted from the Git logs); 2) One developer might use several different identities to author commits. Therefore, we employed the following process to map GitHub accounts to their corresponding identities. For each of the 728 developers included in our study, we first tried to match their GitHub account to the author names in the commits of the repositories they contributed to through PRs. As a result, 360 GitHub usernames could be matched to the commit author names, while no link could be established for the remaining 368 accounts. For this latter, we manually checked their GitHub profile and tried to match their displayed name and email to the author names and emails in Git logs. If no match was found, we manually inspected the "contributors" page of their corresponding repositories on GitHub to check if the developer has made any commits. If the developer did not appear in the list of contributors, we assume no commit was made by the developer. Otherwise, we manually browsed developers' commits to those repositories (which is not possible to retrieve with the GitHub API), and obtained the commit hash. Then, in the local repository, we checked the commit information linked to the commit hash, such that we could obtain the author names they used for commits. As developers might use multiple author names in the commits, we also recorded the other author names associated with the same email addresses they used, and iterated this process with the newly found author names until no new author name emerged. This process was performed by the second author. Through this manual process, we managed to collect the identities of 715 developers, while for the rest 13 we assume they did not make any commit.

3.3.2 Contribution quality measures. We assume that with the knowledge transfer one of the major benefits developers receive is the improvement of the quality of their contributions (*i.e.,* PRs) over time. While there are a few existing metrics to evaluate code quality (see *e.g.,* CK metrics [38] and bug count [28]), some limitations hinder their applications in our study context: 1) The software repositories involved can be written in different programming languages, making it impossible to set universal thresholds for CK metrics, let alone not all programming languages are object-oriented. 2) Metrics like bug count rely on the assumption that bugs can be identified thanks to the consistent usage of issue tracking systems, which is not always the case.We do not pick repositories of specific languages or programming domains as we believe knowledge gained from different types of projects can still be beneficial. In our study we adopt quality contribution measures which are independent from the programming language and application domain. For each submitted PR, we use the following contribution quality measures as dependent variables:

General comments received. The number of general comments received from all the developers other than the PR author. We expect that with the increase of past reviewed PRs (*i.e.,* with more knowledge transfer the developer benefited of), fewer discussions will be triggered by the PR, leading to a reduction of general comments.

Source code comments received. The number of source code comments received from all the developers other than the PR author. Similarly to general comments received, we would expect that the *source code comments received* will decrease over time as well.

Acceptance Rate. The rate of the past PRs acceptance. We expect that the percentage of accepted PRs over time will increase.

Accepted PR closing time. The time (expressed in minutes) between the creation and the closing of the accepted PRs. We expect that the time needed to accept PRs will decrease over time.

Sentiment of source code comments. The sentiment polarity of all source code comments in the PRs. We expect that with the increase of contribution quality more appreciation will be received in the code review. Thus, the sentiment of the developer embedded in the comments should be increasingly positive over time.

Sentiment of general comments. The sentiment polarity of all the general comments in the PRs. Similarly to source code comments, we expect general comments will also be more positive over time.

Sentiment analysis. To calculate the sentiment polarity of the comments in the PRs, we adopted `SentiStrength-SE` [19] and `Senti4SD` [11]. Both tools are designed to work on software-related datasets. For each PR, we aggregate all comments and feed them into these two sentiment analysis tools. Comments are not considered if 1) they are empty, which is possible in general comments when the reviewer just assigns a status to the PR (*e.g.,* "Approved"); or 2) the text contains special characters other than English letters, numbers, punctuation, or emoticons.

`SentiStrength-SE` returns a negative sentiment score (from -1 to -5) and a sentiment score (from +1 to +5). We summed up the two scores and standardized the result in the following way, as suggested by the original authors:

(1) a new score "-1" is assigned if the sum is lower than -1;
(2) a new score "0" is assigned if the sum is in [-1; 1];
(3) a new score "1" is assigned if the sum is higher than 1.

`Senti4SD` returns three sentiment polarity categories (*i.e.,* "positive", "negative" or "neutral"), and we standardized these values to "-1", "0", and "1", respectively.

3.4 Data Analysis

Our hypothesis suggests that developers, who benefited of higher knowledge transfer thanks to the past reviewed PRs they submitted, are also the ones contributing higher quality PRs in the project. We verify this hypothesis thanks to the data previously extracted: Each peer-reviewed PR$_i$ submitted by any of the studied developers represents a row in our dataset, reporting *(i)* the *knowledge transfer measures*, meaning the number of past reviewed PRs performed by the developer before PR$_i$ as well as our control variable, represented by the number of commits she performed in the past (*i.e.,* before

PR$_i$); and *(ii)* the *contribution quality measures* (*i.e.,* acceptance of PRs, number of general comments, etc.). However, the *contribution quality measures* cannot be only computed for the current PR. Indeed, this would make our analysis heavily biased by outliers. For example, a developer having a certain level of *knowledge transfer measures* may have submitted nine PRs before PR$_i$, having all of them accepted but PR$_i$. Indicating a 90% acceptance rate as a proxy for the quality of her recent contributions would be more representative of the actual facts rather than reporting a 0% since only considering PR$_i$. Therefore, we rely on a *fixed sliding window with a length of five PRs* to compute the *contribution quality measures* for each row in our dataset. Instead of reporting the *contribution quality measures* only for PR$_i$, we compute these measures on the most recent five PRs (including PR$_i$) submitted by PR$_i$'s author. There are two exceptions to this process. First, for the measure *accepted PR closing time* we consider the most recent five *accepted* PRs. Second, for the sentiment polarity, we only considered the comments in PR$_i$, since there is a guarantee that PR$_i$ contains at least one comment. We ignore the history of each developer before she performed at least five PRs. This ensures that there are always five PRs falling into the *fixed sliding window.*

Following the above-described process, we created two different datasets, named *cross-project scenario* and *single-project scenario.* In the first, we consider all PRs and all commits performed across all repositories to which a developer contributed, assuming that knowledge acquired thanks to the code review process performed on project P_x, can help developers in submitting better contributions not only to project P_x, but also to project P_y. While both datasets contain one row for each PR performed by the developer in any repository, they differ in the way we compute the *knowledge transfer measures* and the *contribution quality measures.* Given a row in the dataset representing the PR$_i$, in the *single-project scenario* only PRs and commits performed in the past by the developer in the same project PR$_i$ belongs to are considered. This means, for example, that a developer who made 50 PRs in the past, only 12 of which belong to the same project as PR$_i$, will get 12 as the number of past reviewed PRs she submitted in the row corresponding to PR$_i$. Differently, in the *cross-project scenario*, these measures are computed by considering all PRs and commits submitted in any project by PR$_i$'s developer (50 in the example).

Once the datasets were created, we split their rows (*i.e.,* contributions representing PRs) based on the *knowledge transfer measures* of the developer who submitted them. In particular, we extract the first (Q1), second (Q2), and third (Q3) quartile of the distributions for the *number of past reviewed PRs submitted* and the *number of past commits* performed by developers. Then, we split the rows into four groups based on the *number of past reviewed PRs submitted*: *low* (≤ Q1), *medium-low* (> Q1 & ≤ Q2), *medium-high* (> Q2 & ≤ Q3), and *high* (> Q3). Note that, while a contribution (*i.e.,* a row in our dataset) can only appear in one of these groups, the PRs submitted by a developer can appear in more than one group, since her *number of past reviewed PRs submitted* increases over time. We perform the same grouping also for the *number of past commits.* Table 1 lists the value ranges of each "knowledge" measure (the value denoted by *n*) for each group in both *cross-project* and *single-project* scenarios. For example, when we are considering the single project scenario and the knowledge measure # past reviewed PRs, all the PRs whose author made up to eleven PRs in the past fall into the *low experience group.*

Table 1: Groups for each "knowledge" measure

Knowledge measure	Knowledge group	Study scenario	
		Single project	*Cross project*
# past reviewed PRs	low	n≤11	n≤19
	median-low	11<n≤26	19<n≤46
	median-high	26<n≤64	46<n≤110
	high	n>64	n>110
# past commits	low	n≤20	n≤52
	median-low	20<n≤67	52<n≤171
	median-high	67<n≤215	171<n≤446
	high	n>215	n>446

3.4.1 *Statistical methods.*

For both *cross-project* and *single-project* scenarios and each of the experience measures (*i.e.,* # past reviewed PRs, # past commits), we compare via box plots the contribution quality measures in different knowledge groups. The comparisons are also performed via the Mann-Whitney test [14], with results intended as statistically significant at $\alpha = 0.05$. We use the Mann-Whitney test because it is a robust non-parametric test and we did not know *a priori* (and we could not assume) what kind of distribution of data we had [27]. To control the impact of multiple pairwise comparisons (*e.g.,* the "*low knowledge group*" is compared with all the other three groups), we adjust *p*-values with Holm's correction [18]. We estimate the magnitude of the differences by using the Cliff's Delta (*d*), a non-parametric effect size measure. We follow well-established guidelines to interpret the effect size: negligible for $|d| < 0.148$, small for $0.148 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [17].

Note that, before running the above-described analyses, we first remove outliers from the compared data distributions. Given Q1 and Q3 the first and third quartile of a given distribution, and IQR the interquartile range computed as Q3-Q1, we remove all values lower than Q1-(1.5×IQR) or higher than Q3+(1.5×IQR)[40]. This was done for the analyses carried out for *(i)* the number of *general comments received*, *(u)* the number of *source code comments received*, and *(iii)* the *accepted PR closing time.* This was instead not needed for the percentage of accepted PRs (as it is always between 0 and 1), and for the comment sentiment scores (always between -1 and 1).

4 RESULTS

The box plots in Figures 1, 2, 3, and 4 show the trends of the dependent variables (*i.e.,* the *contribution quality measures*), for both the cross- (left) and single- (right) project scenarios, with respect to the two independent variables (*i.e., the knowledge measures*).
In particular, the top part of each figure reports the results obtained when splitting developers into "knowledge groups" based on the past reviewed PRs they submitted, while the bottom part shows the same results when grouping developers based on the number of past commits they performed. The red dot represents the mean value in each box plot.
In Table 2, we report the results of the Mann-Whitney test and Cliff's Delta for past reviewed PRs in the cross-project scenario. The same analyses are reported in Tables 3 (cross-project) and 4 (single-project) for past commits. Due to lack of space, the tables only report results of comparisons that are *(i)* statistically significant (*i.e.,* adjusted p-value lower than 0.05), and *(ii)* have at least a small effect

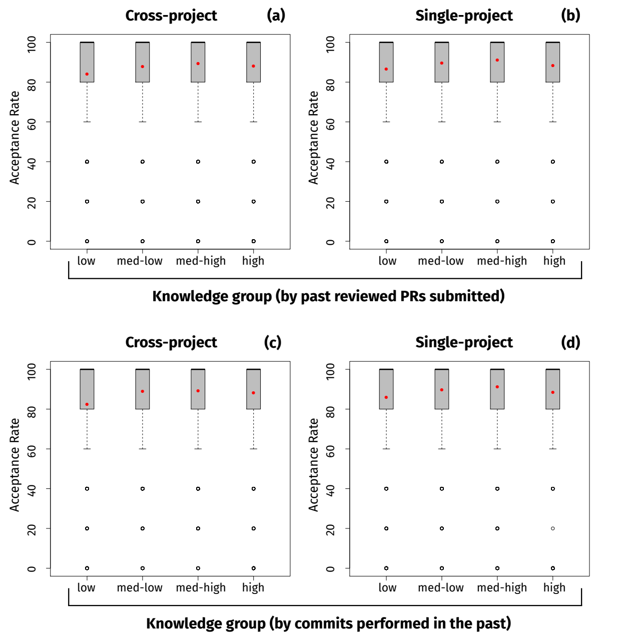

Figure 1: Acceptance rate for PRs submitted by developers.

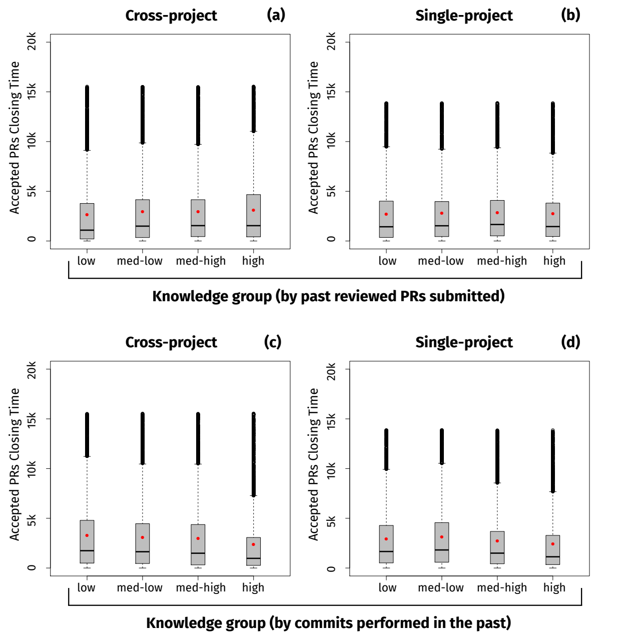

Figure 2: Closing time (in minutes) for PRs submitted by developers.

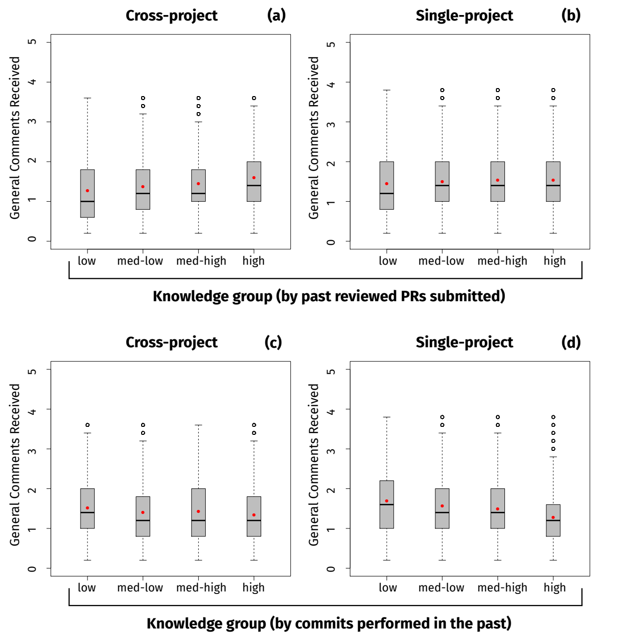

Figure 3: Number of general comments for PRs submitted by developers.

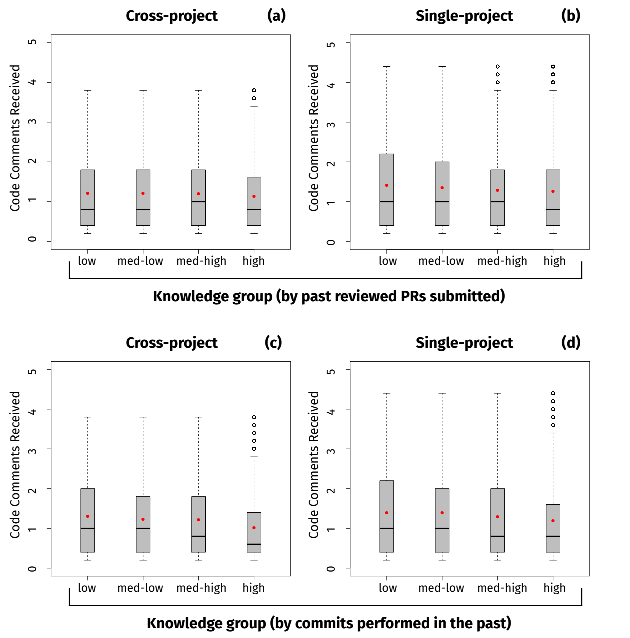

Figure 4: Number of source code comments for PRs submitted by developers.

size (*i.e.,* Cliff's $|d| \geq 0.148$). For the same reason, the table reporting the results achieved in the single-project scenario when using past reviewed PRs as independent variable is not reported, since all comparisons where either not significant or with a negligible effect size. Tables reporting the complete results of the statistical analyses are available in our replication package [12].

In the following, we discuss the achieved results grouping them by dependent variable, commenting the results obtained when using both the past PRs and the past number of commits as criteria to split developers into "knowledge groups".

Table 2: Cross-project scenario - Knowledge groups created by past PRs: Results of the Mann-Whitney test (adj. p-value) and Cliff's Delta (d). We only report results of comparisons that are: *(i)* statistically significant and *(ii)* have at least a *small* effect size.

Test	adj. p-value	d
Acceptance Rate		
No significant differences with at least small d		
Accepted PR Closing Time		
No significant differences with at least small d		
General Comments Received		
low vs medium-high	$<$**0.01**	-0.15 (Small)
low vs high	$<$**0.01**	-0.27 (Small)
medium-low vs high	$<$**0.01**	-0.19 (Small)
Source Code Comments Received		
No significant differences with at least small d		
Sentiment Analysis on General Comments: SentiStrength-SE		
No significant differences with at least small d		
Sentiment Analysis on General Comments: Senti4SD		
No significant differences with at least small d		
Sentiment Analysis on Code Comments: SentiStrength-SE		
No significant differences with at least small d		
Sentiment Analysis on Code Comments: Senti4SD		
No significant differences with at least small d		

4.1 PRs Acceptance Rate

By looking at the boxplots reported in Fig. 1 (a) and (b), we can observe an almost flat trend of the *Acceptance Rate* (expressed in percentage) of PRs when the past reviewed PRs submitted by a developer serve as a proxy for her knowledge. That is, at least by looking at Fig. 1 (top part), we did not observe any effect of the knowledge transfer on the likelihood of future PRs to be accepted.

Looking at the results of the statistical tests, we can also observe that none of the performed comparisons have at least a small effect size (see Table 2).

Concerning our "control variable," meaning the number of commits, we achieved a slight different result: significant differences (with at least a small effect size) can be observed between the knowledge groups (see Table 3). However, this only holds: 1) in the cross-project scenario (no such differences are observed in the single-project setting), and 2) when comparing the *low* group with the top two groups (*i.e., medium-high* and *high*), as well as comparing *medium-low* and *high*. Actually, the effect of the experience acquired through commits over time seems to have an imperceptibly higher impact on the acceptance rate of future PRs as compared to the experience gained through past PRs (compare top and bottom part of Fig. 1).

To summarize, we do not observe any apparent positive impact of the past reviewed PRs submitted by a developer on the likelihood that her future PRs will be accepted (contradicting some previous findings in the literature, *e.g.,* [13, 32, 36, 39]). Note, however, that we adopted a completely different experimental design, and we only considered past reviewed PRs as independent variable.

Instead, developers are more likely to improve their PR acceptance along with the increase of their committing experience (as observed through the commits-based analysis).

4.2 Accepted PRs Closing Time

As for the *accepted PR closing time*, the top part of Fig. 2 is also quite flat, for both cross- and single-project scenarios. This finding is also supported by the results of the statistical analysis, reporting *negligible* effect sizes for all performed comparisons.

Such a result was quite surprising for us, since we expected that the higher the knowledge acquired by developers through PRs, the lower the closing time of their accepted PRs. While we do not have any empirical evidence to explain the lack of such a trend, one possibility is that more experienced developers are responsible for more complex PRs, that require longer reviewing time thus "nullifying" the advantage brought by the acquired knowledge. Such a finding would be in line with what discussed by Zeller in his book *Why Programs Fail* [42], in which the author reports that Erich Gamma, the master developer of Eclipse, was the second most defect-prone Eclipse developer. The explanation for such a finding was indeed that more experienced developers tend to perform more complex and critical tasks [42].

When performing the same analysis for the *past commits* independent variable (bottom part of Fig. 2), we observe a slight decrease of reviewing time when moving from the *low* towards the *high* group in the cross-project scenario, with the statistical tests reporting a significant difference with a non-negligible effect size only when comparing the *low* and the *high* groups (see Table 3).

4.3 Comments Posted in PRs

We discuss together our findings for both the number of *general comments* (Fig. 3) and *source code comments* (Fig. 4) posted in the PRs submitted by different groups of developers. We first focus on the top part of both figures (*i.e.,* results related to the past reviewed PRs).

These two figures together tell an interesting story. While developers who acquired more knowledge over time receive more general comments (possibly indicating the higher complexity of the changes they implement), the number of *source code comments*, meaning specific recommendations on how to improve the code, does not increase with the increase of the knowledge. This means that, despite the PRs submitted by developers who performed a higher number of reviewed PRs in the past are discussed more, they do not receive a higher number of comments for source code. This is also confirmed by the statistical tests for the cross-project scenario (see Table 2), with: 1) significant differences observed for the number of general comments received in the *low* and *medium-low* groups when compared with the *high* group, as well as in the *low* group when compared with the *medium-low* group, and 2) no differences found for what concerns the number of received code comments among the different groups.

When looking at the commits-based analysis (bottom part of Figures 3 and 4), significant differences with a small effect size can be observed regarding the number of general comments received when comparing the *high* group to all other groups (see Table 4) in single-project scenario. Meanwhile, similar differences can also be found when comparing the source code comments received between the *low* group and the *high* group in cross-project scenario.

Overall, the comments posted during the PRs reviewing process seem to be the only dependent variable in our study for which

Table 3: Cross-project scenario - Knowledge groups created by past commits: Results of Mann-Whitney test (adj. p-value) and Cliff's Delta (d). We only report results of comparisons that are *(i)* statistically significant, and *(ii)* have at least a *small* effect size.

Test	adj. p-value	d
Acceptance Rate		
low vs medium-low	**<0.01**	-0.16 (Small)
low vs medium-high	**<0.01**	-0.16 (Small)
low vs high	**<0.01**	-0.21 (Small)
Accepted PR Closing Time		
low vs high	**<0.01**	0.17 (Small)
General Comments Received		
No significant differences with at least small d		
Source Code Comments Received		
low vs high	**<0.01**	0.16 (Small)
Sentiment Analysis on General Comments: SentiStrength-SE		
low vs high SSE	**<0.01**	0.16 (Small)
Sentiment Analysis on General Comments: Senti4SD		
low vs high 4SD	**<0.01**	0.17 (Small)
Sentiment Analysis on Code Comments: SentiStrength-SE		
No significant differences with at least small d		
Sentiment Analysis on Code Comments: Senti4SD		
No significant differences with at least small d		

we observed some possible positive influence of the knowledge acquired in the code review process. Indeed, while PRs submitted by more experienced developers (in terms of reviewed PRs they submitted in the past) are more discussed, they do not receive more requests for code changes. Such an effect is also visible when using the past commits as independent variable in single-project setting.

4.4 Sentiment Polarity of Comments

As far as the Sentiment Polarity is concerned, we do not show any box plot for space reason (they are available in our replication package [12]). However, the results of the statistical tests are reported in the Tables 2 (cross-project, past PRs), 3 (cross-project, past commits), and 4 (local-project, past commits). As previously said, no results are reported for the local-project scenario when using past PRs due to the non-significant p-values and/or negligible d effect size achieved in all comparisons.

We found that neither positive nor negative polarities in the source code discussions prevail in both the cross and single-project studies. Such an outcome is plausible due to the fact that code review discussions mostly concern topics like *(i)* defect detecting, *(ii)* reviewer assigning, *(iii)* contribution encouraging, and so on [22]. Second, only the comparison of sentiment polarity in general comments between the *low* group and the *high* group provides a significant result (*i.e.,* the two extremes, with "newcomers" and very experienced developers). In this case, we found that the sentiment polarity is generally higher in discussions related to PRs opened by developers in the *low* group in the cross-project scenario. This may be due to the fact that reviewers tend to be more positive with newcomers to not discourage them in contributing again in the future. Note that the findings related to the sentiment polarity of comments are confirmed by both sentiment analysis tools used in our study.

Table 4: Single-project scenario - Knowledge groups created by past commits: Results of Mann-Whitney test (adj. p-value) and Cliff's Delta (d). We only report results of comparisons that are *(i)* statistical significant, and *(ii)* have at least a *small* effect size.

Test	adj. p-value	d
Acceptance Rate		
No significant differences with at least *small* d		
Accepted PR Closing Time		
No significant differences with at least *small* d		
General Comments Received		
low vs high	**<0.01**	0.31 (Small)
medium-low vs high	**<0.01**	0.21 (Small)
medium-high vs high	**<0.01**	0.16 (Small)
Source Code Comments Received		
No significant differences with at least *small* d		
Sentiment Analysis on General Comments: SentiStrength-SE		
No significant differences with at least *small* d		
Sentiment Analysis on General Comments: Senti4SD		
No significant differences with at least *small* d		
Sentiment Analysis on Code Comments: SentiStrength-SE		
No significant differences with at least *small* d		
Sentiment Analysis on Code Comments: Senti4SD		
No significant differences with at least *small* d		

4.5 Answering our Research Question

Our study led to what we can define a negative result. For most of the analyzed dependent variables we did not find any strong impact of the knowledge transfer in the code review process on the quality of the contributions submitted by developers in open source projects. In particular, for the *PRs acceptance rate*, we did not observe any positive effect in the cross-project scenario when using past PRs as a proxy for knowledge transfer. Instead, an increase of experience over time might be more important for the improvement of the *PRs acceptance rate*, as demonstrated by the results achieved when using past commits as independent variable.

For the *closing time of accepted PRs*, most of the times we found no impact of the knowledge acquired in past PRs. As said, this may be due to the fact that more experienced developers tend to submit more complex PRs that, in some way, nullify the shorter reviewing time they would benefit of otherwise. Additional investigations are needed to understand the reasons behind such a result.

The *comments posted in PRs* are the only dependent variables for which we observed some influence of the knowledge acquired in past reviewed PRs. Indeed, while the PRs submitted by developers in the *high* group are generally more discussed, they receive a similar amount of recommendations for improving the contributed code, indicating a higher quality of the submitted PR. Also, such a phenomenon was not observable when using commits as independent variable in the cross-project scenario. Finally, no major differences were observed in the polarity of sentiments for comments posted in PRs submitted by developers having different levels of knowledge as assessed by both past PRs and past commits.

Overall, our findings failed to provide some quantitative evidence about the benefits brought by a code review process in improving developers' skills over time. The reasons behind such a result certainly deserve additional investigation, since knowledge transfer is one of the main motivations for modern code review. We believe that different experiments, using different experimental designs (*e.g.,* different dependent and independent variables) are needed to corroborate or contradict our findings.

5 THREATS TO VALIDITY

To comprehend the strengths and limitations of our study, the threats that could affect the results and their generalization are presented and discussed here. Despite our efforts to mitigate as many threats to validity as possible, some are still unavoidable.

Threats to *construct validity* concern the relation between the theory and the observation, and in this work are mainly due to the measurements we performed:

- *The way in which we measured knowledge transfer in code review.* There are no accepted metrics to quantitatively assess the notion of knowledge transfer, especially in a context, such as that of mining software repositories, in which there is no direct access to the studied developers. We assumed that the number of past reviewed PRs, that have been submitted by a developer, represent a good proxy of the knowledge transfer that developer has benefited of. To at least mitigate the threat represented by such an assumption, we only considered past PRs that actually received at least one comment by a peer (non-bot) developer. This should at least ensure that a review process was actually carried out for the considered PRs. These measures may not precisely capture the knowledge transfer process given its complex nature. On the basis of our study (design and outcomes), additional investigations are needed to understand which quantitative proxies can best quantify the knowledge gained during code review process.
- *The measures used to assess the quality of contributions over time.* We adopt a number of indicators that should reasonably be related to the quality of the contributions submitted by a developer via PRs. For example, we assumed that a higher acceptance rate of the submitted PRs is related to higher quality contributions. While such assumption might look reasonable, there might be corner cases in which they do not hold, *e.g.,* PRs accepted despite the fact that they provide a sub-optimal solution, maybe due to the need for fixing, at least partially, a blocking bug. Also, one of our measures (*i.e.,* closing time of accepted PRs) is based on time-related aspects that, when mined from software repositories, can bring noise to the performed measurements. Indeed, there is no guarantee that a review process started right after the PR submission. Thus, longer/shorter reviewing times might be due to factors completely unrelated to the quality/complexity of the submitted contribution.
- *The approach for mapping GitHub user names to commit author names.* There is still a possibility that some developers might use identities we did not discover, or intentionally hide their identities when authoring commits. However, by iterative linking process and manual inspection, we believe the impact has been limited to the possible minimum.
- *The sentiment polarity assessment provided by sentiment analysis tools.* Previous studies showed that state-of-the-art sentiment analysis tools provide poor performance when used in context different from the ones they have been designed for [23]. Both tools we adopted [11, 19] have been designed to work on software-related data. However, they have been experimented on different datasets as compared to the one used in this paper and, as a consequence, their performance on the PR comments can be different from the one reported in the original papers.

Threats to *internal validity* concern external factors we did not consider that could affect the variables and the relations being investigated. The differences observed between the groups of developers we created may be due to several confounding factors (*e.g.,* developers performing more PRs acquire more skills over time not due to the code review process, but thanks to the accumulated experience). For this reason, we also replicated our analyses by using the number of past commits to split the developers into "knowledge groups". This helped, for example, to provide a better interpretation of the results achieved for the *PRs acceptance rate* independent variable.

Threats to *conclusion validity* concern the relation between the treatment and the outcome. Wherever necessary, we used suitable statistical inferences to support our conclusions: we used the Mann-Whitney test (with adjusted p-values due to multiple comparisons) and Cliff's d effect size.

Threats to *external validity* concern the generalizability of our findings. We tried to achieve high generalizability by considering the complete contribution history of 728 developers, for a total of 32,062 PRs spanning 4,981 repositories. Also, we did not apply any filter related to the programming language, since all the steps of our study are language-independent.

6 CONCLUSIONS

We presented a quantitative study to investigate knowledge transfer in code review. Our results were mostly negative: we were not able to capture the positive role played by code review in knowledge transfer among developers, as was previously suggested in the literature [2].This came to us as a surprise, as we were confident to see at least significant traces of the knowledge transfer, because despite not supporting the findings of Bacchelli and Bird [2] given our results, we actually are convinced that their claims are correct. This raises a number of questions that we have addressed in part throughout the latter part of the paper, where we conjecture possible fallacies in our experiment design and notable threats to validity that are difficult to fully address, especially those regarding the measures we used to quantify the impact of knowledge transfer.

We stress the fact that our findings do not contradict previous qualitative results reported in the literature, but rather call for additional investigations aimed at understanding how (and if) we can actually capture the knowledge transfer in code review in a quantitative way. Therefore, our main direction for future work includes additional studies investigating the same research questions with a different experimental design. Specifically, we will investigate which measures can be used as a precise proxy to represent the knowledge transfer, in both quantitative and qualitative way. The data used in our study is publicly available [12].

ACKNOWLEDGMENTS

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects PROBE (SNF Project No. 172799) and CCQR (SNF Project No. 175513), and CHOOSE for sponsoring our trip to the conference.

REFERENCES

[1] E. M. Awad and H. M. Ghaziri. 2009. Knowledge Management. International Edition.

[2] A. Bacchelli and C. Bird. 2013. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. 712–721.

[3] T. Baum, O. Liskin, K. Niklas, and K. Schneider. 2016. A Faceted Classification Scheme for Change-Based Industrial Code Review Processes. In *Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS' 16)*. 74–85.

[4] G. Bavota and B. Russo. 2015. Four eyes are better than two: On the impact of code reviews on software quality. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME '15)*. 81–90. https://doi.org/10.1109/ICSM.2015.7332454

[5] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey. 2013. The influence of non-technical factors on code review. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE 2013)*. 122–131.

[6] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens. 2014. Modern Code Reviews in Open-source Projects: Which Problems Do They Fix?. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. ACM, New York, NY, USA, 202–211.

[7] M. Bernhart and T. Grechenig. 2013. On the understanding of programs with continuous code reviews. In *Proceedings of the 21st International Conference on Program Comprehension (ICPC '13)*. 192–198. https://doi.org/10.1109/ICPC.2013.6613847

[8] M. Bernhart, A. Mauczka, and T. Grechenig. 2010. Adopting Code Reviews for Agile Software Development. In *2010 Agile Conference*. 44–47. https://doi.org/10.1109/AGILE.2010.18

[9] F. Bjørnson and T. Dingsøyr. 2008. Knowledge management in software engineering: A systematic review of studied concepts, findings and research methods used. *Information and Software Technology* 50, 11 (2008), 1055–1068.

[10] A. Bosu, M. Greiler, and C. Bird. 2015. Characteristics of Useful Code Reviews: An Empirical Study at Microsoft. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15)*. 146–156. https://doi.org/10.1109/MSR.2015.21

[11] F. Calefato, F. Lanubile, F. Maiorano, and N. Novielli. 2018. Sentiment Polarity Detection for Software Development. *Empirical Software Engineering* 23, 3 (June 2018), 1352–1382. https://doi.org/10.1007/s10664-017-9546-9

[12] M. Caulo, B. Lin, G. Bavota, G. Scanniello, and M. Lanza. 2020. https://tinyurl.com/wn9my8f.

[13] D. Chen, K. Stolee, and T. Menzies. 2019. Replication Can Improve Prior Results: A GitHub Study of Pull Request Acceptance. In *Proceedings of the 27th International Conference on Program Comprehension (ICPC '19)*. 179–190. https://doi.org/10.1109/ICPC.2019.00037

[14] W. J. Conover. 1999. Practical nonparametric statistics. (1999).

[15] J. Czerwonka, M. Greiler, and J. Tilford. 2015. Code Reviews Do Not Find Bugs: How the Current Code Review Best Practice Slows Us Down. In *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*. IEEE Press, 27–28.

[16] G. Destefanis, M. Ortu, D. Bowes, M. Marchesi, and R. Tonelli. 2018. On Measuring Affects of Github Issues' Commenters. In *Proceedings of the 3rd International Workshop on Emotion Awareness in Software Engineering (SEmotion '18)*. Association for Computing Machinery, New York, NY, USA, 14–19. https://doi.org/10.1145/3194932.3194936

[17] R. J. Grissom and J. J. Kim. 2005. Effect sizes for research: A broad practical approach. *Mahwah, NJ: Earlbaum* (2005).

[18] S. Holm. 1979. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics* (1979), 65–70.

[19] M. Islam and M. Zibran. 2018. SentiStrength-SE: Exploiting Domain Specificity for Improved Sentiment Analysis in Software Engineering Text. *Journal of Systems and Software* 145 (08 2018). https://doi.org/10.1016/j.jss.2018.08.030

[20] C. F. Kemerer and M. C. Paulk. 2009. The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data. *Software Engineering, IEEE Transactions on* 35, 4 (2009), 534–550.

[21] O. Kononenko, T. Rose, O. Baysal, M. Godfrey, D. Theisen, and B. de Water. 2018. Studying Pull Request Merges: A Case Study of Shopify's Active Merchant. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP '18)*. 124–133.

[22] Z. Li, Y. Yu, G. Yin, T. Wang, and H.-M. Wang. 2017. What Are They Talking About? Analyzing Code Reviews in Pull-Based Development Model. *Journal of Computer Science and Technology* 32 (11 2017), 1060–1075. https://doi.org/10.1007/s11390-017-1783-2

[23] B. Lin, F. Zampetti, G. Bavota, M. Di Penta, M. Lanza, and R. Oliveto. 2018. Sentiment analysis for software engineering: how far can we go?. In *Proceedings of the 40th International Conference on Software Engineering, (ICSE' 18)*. 94–104.

[24] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. 2014. The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the Qt, VTK, and ITK Projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. 192–201.

[25] R. Morales, S. McIntosh, and F. Khomh. 2015. Do Code Review Practices Impact Design Quality? A Case Study of the Qt, VTK, and ITK Projects. In *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER '15)*. 171–180.

[26] R. Morales, S. McIntosh, and F. Khomh. 2015. Do code review practices impact design quality? A case study of the Qt, VTK, and ITK projects. *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER '15)* (04 2015), 171–180. https://doi.org/10.1109/SANER.2015.7081827

[27] H. Motulsky. 2010. *Intuitive biostatistics: a non-mathematical guide to statistical thinking*. Oxford University Press. http://books.google.it/books?id=R477U5bAZs4C

[28] Alessandro Murgia, Giulio Concas, Sandro Pinna, Roberto Tonelli, and Ivana Turnu. 2009. Empirical study of software quality evolution in open source projects using agile practices. In *Proc. of the 1st International Symposium on Emerging Trends in Software Metrics*. 11.

[29] M. V. Mäntylä and C. Lassenius. 2009. What Types of Defects Are Really Discovered in Code Reviews? *IEEE Transactions on Software Engineering* 35, 3 (May 2009), 430–448. https://doi.org/10.1109/TSE.2008.71

[30] M. Ortu, M. Marchesi, and R. Tonelli. 2019. Empirical Analysis of Affect of Merged Issues on GitHub. In *Proceedings of the 4th International Workshop on Emotion Awareness in Software Engineering (SEmotion '19)*. 46–48. https://doi.org/10.1109/SEmotion.2019.00017

[31] P. Pooput and P. Muenchaisri. 2018. Finding Impact Factors for Rejection of Pull Requests on GitHub. In *Proceedings of the 2018 VII International Conference on Network, Communication and Computing (ICNCC 2018)*. 70–76.

[32] M. M. Rahman and C. Roy. 2014. An Insight into the Pull Request of GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR '14)*. 364–367. https://doi.org/10.1145/2597073.2597121

[33] P. C. Rigby, D. M. German, and M.-A. Storey. 2008. Open Source Software Peer Review Practices: A Case Study of the Apache Server. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. 541–550.

[34] P. C. Rigby and M.-A. Storey. 2011. Understanding Broadcast Based Peer Review on Open Source Software Projects. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. 541–550.

[35] M. Silva, M. Valente, and R. Terra. 2016. Does Technical Debt Lead to the Rejection of Pull Requests?. In *Proceedings of the XII Brazilian Symposium on Information Systems on Brazilian Symposium on Information Systems: Information Systems in the Cloud Computing Era-Volume 1*. 248–254.

[36] D. Soares, M. de Lima Júnior, L. Murta, and A. Plastino. 2015. Acceptance Factors of Pull Requests in Open-Source Projects. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC '15)*. Association for Computing Machinery, New York, NY, USA, 1541–1546. https://doi.org/10.1145/2695664.2695856

[37] D. M. Soares, M. L. d. L. Júnior, L. Murta, and A. Plastino. 2015. Rejection Factors of Pull Requests Filed by Core Team Developers in Software Projects with High Acceptance Rates. In *Proceedings of the 14th International Conference on Machine Learning and Applications (ICMLA '15)*. 960–965. https://doi.org/10.1109/ICMLA.2015.41

[38] Ramanath Subramanyam and Mayuram S. Krishnan. 2003. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on software engineering* 29, 4 (2003), 297–310.

[39] J. Tsay, L. Dabbish, and J. Herbsleb. 2014. Influence of Social and Technical Factors for Evaluating Contribution in GitHub. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*. Association for Computing Machinery, New York, NY, USA, 356–366. https://doi.org/10.1145/2568225.2568315

[40] J. W. Tukey. 1977. *Exploratory Data Analysis*. Addison-Wesley.

[41] P. Weißgerber, D. Neu, and S. Diehl. 2008. Small Patches Get in!. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories (MSR '08)*. 67–76.

[42] A. Zeller. 2005. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.