The Review Linkage Graph for Code Review Analytics

A Recovery Approach and Empirical Study

Toshiki Hirao
Nara Institute of Science and Technology
Nara, Japan
hirao.toshiki.ho7@ais.naist.jp

Shane McIntosh
McGill University
Montréal, Canada
shane.mcintosh@mcgill.ca

Akinori Ihara
Wakayama University
Wakayama, Japan
ihara@sys.wakayama-u.ac.jp

Kenichi Matsumoto
Nara Institute of Science and Technology
Nara, Japan
matumoto@is.naist.jp

ABSTRACT

Modern Code Review (MCR) is a pillar of contemporary quality assurance approaches, where developers discuss and improve code changes prior to integration. Since review interactions (e.g., comments, revisions) are archived, analytics approaches like reviewer recommendation and review outcome prediction have been proposed to support the MCR process. These approaches assume that reviews evolve and are adjudicated independently; yet in practice, reviews can be interdependent.

In this paper, we set out to better understand the impact of review linkage on code review analytics. To do so, we extract review linkage graphs where nodes represent reviews, while edges represent recovered links between reviews. Through a quantitative analysis of six software communities, we observe that (a) linked reviews occur regularly, with linked review rates of 25% in OpenStack, 17% in Chromium, and 3%–8% in Android, Qt, Eclipse, and Libreoffice; and (b) linkage has become more prevalent over time. Through qualitative analysis, we discover that links span 16 types that belong to five categories. To automate link category recovery, we train classifiers to label links according to the surrounding document content. Those classifiers achieve F1-scores of 0.71–0.79, at least doubling the F1-scores of a ZeroR baseline. Finally, we show that the F1-scores of reviewer recommenders can be improved by 37%–88% (5–14 percentage points) by incorporating information from linked reviews that is available at prediction time. Indeed, review linkage should be exploited by future code review analytics.

CCS CONCEPTS

• **Software and its engineering** → *Software evolution.*

KEYWORDS

Code review, software analytics, mining software repositories

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338949

ACM Reference Format:
Toshiki Hirao, Shane McIntosh, Akinori Ihara, and Kenichi Matsumoto. 2019. The Review Linkage Graph for Code Review Analytics: A Recovery Approach and Empirical Study. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338906.3338949

1 INTRODUCTION

Modern Code Review (MCR)—a lightweight variant of traditional code inspections [14]—allows developers to discuss the premise, content, and structure of code changes. Many communities adopt a Review-Then-Commit (RTC) philosophy, where each code change must satisfy review-based criteria before integration into official repositories is permitted. Since MCR tools archive reviewing activities (e.g., patch revisions, review participants, discussion threads), active development communities generate plenty of MCR data.

Researchers have proposed analytics approaches that leverage MCR data to support practitioners. For example, *reviewer recommenders* [4, 31, 42, 44, 48] help developers to select appropriate reviewers and *review outcome predictors* [16, 22, 23] estimate the likelihood of a code change eventually being integrated.

When performing code review analytics, each review has traditionally been treated as an independent observation; yet in practice, reviews may be interdependent. By ignoring connections between reviews, review analytics approaches may underperform. For example, the discussion for Review #314319 (https://review.openstack.org/#/c/314319/) of the OpenStack Neutron project occurred on its linked Review #225995. Without considering the comments on the linked review, a review outcome predictor would mistakenly presume that Review #314319 had no discussion, and would thus be unlikely to be integrated. Moreover, Review #134811 (https://review.openstack.org/#/c/134811/) of the OpenStack Nova project is abandoned because a competing solution in the linked Review #134853 was integrated. To ensure a fair review process [15], reviewer lists of competing solutions may need to be synchronized; yet links are not analyzed by today's reviewer recommenders.

In this paper, we set out to better understand the impact of review linkage on code review analytics. To do so, we extract the *review linkage graph* from six active MCR communities—a directed graph where nodes represent reviews and edges represent links between reviews. We analyze and leverage those graphs to address the following four research questions:

(RQ1) To what degree are reviews linked?
Motivation: To gain an initial intuition about the connectedness of reviews, we first set out to quantitatively analyze the extracted review linkage graphs.
Results: Linkage rates range from 3% to 25%. Linkage tends to be more common in the two largest communities (25% in OpenStack and 17% in Chromium), likely because they have invested more heavily in code reviewing. Indeed, these communities have significantly more comments and reviewers per review (pairwise Mann-Whitney U tests with Bonferroni correction, $\alpha = 0.01$; and non-negligible Cliff's delta effect sizes).

(RQ2) Why are reviews being linked?
Motivation: The potential reasons for review linkage are manifold. To explore these reasons, we set out to qualitatively analyze recovered links between reviews.
Results: Using open coding [12] and card sorting [30], we discover 16 types of review links that belong to five categories, i.e., Patch Dependency, Broader Context, Alternative Solution, Version Control Issues, and Feedback Related. These different link types have different implications for review analytics techniques. For example, while Broader Context links indicate that a discussion may span across linked reviews, Alternative Solution links point out competing solutions.

(RQ3) To what degree can link categories be automatically recovered?
Motivation: The qualitative approach that we used to address RQ2 is not scalable enough for large-scale analyses of link categories. Hence, we want to explore the feasibility of training automatic classifiers to identify link categories.
Results: We train link category classifiers using five classification techniques. These classifiers at least double the performance of a ZeroR baseline, achieving a precision of 0.71–0.77, a recall of 0.72–0.92, and an F1-score of 0.71–0.79.

(RQ4) To what degree do linked reviews impact code review analytics?
Motivation: While RQ1–RQ3 suggest that linkage may impact reviewer analytics, the extent of that impact is unknown. We set out to quantify that impact by comparing prior review analytics techniques [16, 48] to extended versions that are link aware.
Results: Code review analytics tend to underperform on linked reviews. In 41%–84% of linked reviews, reviewer recommenders omit at least one shared reviewer. Moreover, review outcome predictors misclassify 35%–39% of linked reviews. Link-aware approaches improve the F1-score of reviewer recommenders by 37%–88% (5–14 percentage points).

Our empirical study indicates that linkage is a rich activity that should be taken into consideration in future MCR studies and tools. In addition, this paper contributes a replication package,[1] which includes (a) review linkage graphs that feature 1,466,702 reviews and 231,341 links from the six studied communities; (b) 752 manually coded links spanning 16 types and five categories [RQ2]; and (c) scripts that reproduce our statistical analyses [RQ1, RQ3, RQ4].

Paper Organization. The remainder of this paper is organized as follows. Section 2 situates this paper with respect to the related work. Section 3 describes the studied communities and their MCR processes. Sections 4–7 present the experiments that we conducted to address RQ1–RQ4, respectively. Section 8 discusses the broader implications of our results. Section 9 discloses threats to the validity of our study. Finally, Section 10 draws conclusions.

[1] https://github.com/software-rebels/ReviewLinkageGraph

2 RELATED WORK

Linkage of related software artifacts has long been considered an important phenomenon. Canfora *et al.* [11] found that links between issue reports of different software projects are not uncommon. Boisselle and Adams [8] reported that 44% of bug reports in Ubuntu are linked to indicate duplicated work. Ma *et al.* [29] showed that linked issues delay the release cycle and increase maintenance costs. Moreover, they found that recovering a link is a difficult task, often taking developers more than one day to do by hand.

To ease the recovery of links, researchers have proposed automatic approaches. Antoniol *et al.* [2] applied Natural Language Processing (NLP) techniques to detect links between source code and related documents. Alqahtani *et al.* [1] proposed an automatic approach to recover links between API vulnerabilities. Guo *et al.* [17] used deep learning techniques that exploit domain knowledge to detect semantic links. Rath *et al.* [34] detect missing links between commits and issues using process and text-related features.

Linkage also appears in peer code review settings. Zampetti *et al.* [46] found that developers reference other resources in reviews to enhance documentation of pull requests. Perhaps the most similar prior work is that of Li *et al.* [27], who reported that 27% of GitHub pull requests from 16,584 Python projects on GitHub have links, which span six types. This paper expands upon the work of Li *et al.* by studying linkage in six large and successful software communities (rather than a broad sample of GitHub projects), and the impact of linkage on review analytics approaches.

2.1 Reviewer Recommendation

Selecting appropriate reviewers plays an important role in the value that is generated by a code review. Bosu *et al.* [9] found that the reviewer expertise is an important factor in determining whether Microsoft developers consider code review feedback useful. Kononenko *et al.* [25] also found that reviewer expertise is a key factor in developer perceptions of code review quality at Mozilla.

To support authors in selecting appropriate reviewers, researchers have proposed reviewer recommenders. For example, Balachandran [4] proposed ReviewBot, which recommends reviewers based on past contributions to the lines that were modified by the patch. Thongtanunam *et al.* [42] proposed RevFinder, which recommends reviewers based on their prior contributions to files in similar locations within the codebase. More recent work has improved reviewer recommendations by leveraging past review contributions [48], technological experience and experience with other related projects [31], and the content of the patch itself [44]. Kovalenko *et al.* [26] question the value of recommending reviewers in cases where the best reviewers are already known.

Review links may also provide useful information for the reviewer recommenders. Since links may indicate a (strong) relationship between connected reviews, reviewers who are recommended for one of the linked reviews may also need to be recommended

Table 1: An overview of our subject communities.

Product	Language	Studied Period	#Reviews	#Revs	#Projects
OpenStack	Python	09/2011–01/2018	533,050	12,359	1,804
Chromium	JavaScript	04/2011–01/2018	364,079	7,442	410
Android	Java	10/2008–01/2018	229,210	7,614	1,049
Qt	C++	07/2011–01/2018	188,981	2,377	170
Eclipse	Java	04/2012–01/2018	106,515	2,191	380
Libreoffice	Python	04/2012–01/2018	44,867	822	34
Total		31.8 years	1,466,702	32,805	3,847

for the others. For example, Review #109178 is another attempt at tackling the underlying task of (the abandoned) Review #105238.[2] To preserve continuity of the review discussion, the reviewers of Review #105238 should also be recommended for Review #109178.

2.2 Code Review Outcome

Not all changes that are submitted for code review end up being integrated. Indeed, Weißgerber *et al.* [43] found that 39% and 42% of OpenAFS and FLAC code changes were eventually integrated. Jiang *et al.* [23] found that 33% of Linux code reviews were eventually integrated. Baysal *et al.* [5] found that 36%–39% of code changes in Mozilla Firefox were rejected and resubmitted at least once.

To better understand the chances of submissions being integrated, researchers have studied the characteristics of code changes (and their reviews) that were eventually integrated. For example, Weißgerber *et al.* [43] found that small code changes are integrated more frequently than large ones. Jiang *et al.* [23] showed that prior experience, patch maturity, and code churn significantly impact the likelihood of integration. Baysal *et al.* [6, 7] found that non-technical issues are a common reason for abandonment in WebKit and Blink projects. Moreover, Tao *et al.* [41] found that patch design issues like suboptimal solutions and incomplete fixes are often raised during the reviews of the abandoned code changes of the Eclipse and Mozilla projects.

Review links may affect or imply review outcomes. For example, since reviews that supersede prior reviews have had the opportunity to improve the design of the code change, they may have a higher likelihood of being integrated than initial submissions. Moreover, large tasks that have been divided into a series of reviews (e.g., Reviews #102532[3] and #102543[4] of OpenStack) will have review outcomes that are inherently linked.

Prior work suggests that review linkage is important; however, the extent of linkage in large communities and its impact on review analytics is unclear. In this paper, we set out to bridge these gaps.

3 STUDIED COMMUNITIES

The goal of our study is to extract and analyze review graphs of large software communities that have invested in their MCR processes. To do so, we focus on the six popular communities that appear in the recent related work [19, 42, 45]. OpenStack is a cloud computing platform. Chromium is an open source web browser. Android is a mobile software platform. Qt is a cross-platform application framework. Eclipse is an Integrated Development Environment (IDE) and associated tools. Libreoffice is a free and open implementation of the Office software suite.

The studied communities use Gerrit for code review. To conduct our study, we collect MCR data using the Gerrit API. Table 1 provides an overview of the collected data. Below, we provide an overview of the Gerrit processes of the studied communities.

Gerrit Code Review Process. Gerrit is a popular, web-based code review management tool that tightly integrates with the Git version control system. Rather than pushing code changes directly to an upstream Git repository, developers push changes to Gerrit, where only after satisfying project-specific review criteria (e.g., a member of the core team approves the changes) may the author push their changes into the upstream Git repository. Similar to other code reviewing processes (e.g., GitHub pull requests), each Gerrit review goes through the following phases:

(1) **Uploading a proposed set of changes.** An author uploads a proposed set of changes to the Gerrit system and invites reviewers to critique it by leaving comments for the author to clarify, discuss, or address.
(2) **Soliciting peer feedback.** Reviewers critique the premise, content, and structure of the proposed set of changes and provide feedback in the form of *general or inline comments* that correspond to the entire change and lines within it, respectively.
(3) **Revising the proposed patch.** Authors may revise their set of changes to address reviewer feedback. After revising, the review returns to phase 2.
(4) **Automated testing.** Automated tests are executed on each set of changes to mitigate the risk of introducing regression issues. If these tests fail, the set of changes is blocked from integration until the author uploads a revision that addresses the issues. When revisions are uploaded, the review returns to phase 2.
(5) **Final integration.** Once the set of changes satisfies reviewer and automated testing criteria, the author can integrate it into upstream Git repositories.

4 REVIEW GRAPH EXTRACTION (RQ1)

In this section, we set out to study the extent to which reviews are linked to one another in our studied communities. To do so, we extract review graphs from the studied communities. The review graph $RG = (R, L)$ is a directed graph with the following properties:

- Graph nodes R represent review entries. Each review entry $r \in R$ is comprised of a set of properties, such as ID_r (a unique review identifier), D_r (the change description), PR_r (the set of patch revisions), REV_r (the set of reviewers). GC_r and IC_r (general and inline comments, respectively).
- Graph edges L represent links between review entries. Each edge $l \in L$ has a type T_l that describes why the link was recorded, which we study in Sections 5–7.

The set of reviews R is what has typically been extracted and analyzed by prior work on code review. In this section, we first propose a lightweight approach to recover graph edges L from the D_r, GC_r, and IC_r fields of each $r \in R$ (4.1). Then, we apply that approach to the studied communities and analyze the extracted graphs to address RQ1 (4.2).

[2] https://review.openstack.org/#/c/105238/
[3] https://review.openstack.org/#/c/102532/
[4] https://review.openstack.org/#/c/102543/

4.1 Link Recovery Approach

We perform a preliminary analysis of 410 randomly selected reviews to gain insight into the linkage habits within the studied communities. We observe that links appear in review description fields (D_r), as well as within general and inline comment threads (GC_r and IC_r). Moreover, we observe two ways that links are recorded:

(1) By Change ID (e.g., `Ic8aaa0728a43936cd4c6e1ed590e01ba8f0fbf5b`), i.e., a 40-digit hexadecimal identifier assigned to each review at creation time (prefixed with an `I` to avoid confusion with Git commit IDs).
(2) By URL (e.g., `https://review.openstack.org/#/c/1111`).

Thus, to recover links from a review $r \in R$, we scan its description D_r as well as all of the general and inline comments (GC_r, IC_r) using regular expressions. To detect Change IDs, our regular expression scans for terms of the form `I[0-9a-f]{40}`. To detect URLs, our regular expression is of the form `https?://PROJ/#/c/[1-9]+[0-9]*`, where `PROJ` is replaced with the base URL of the project (e.g., `review.openstack.org`). For the Chromium community, our regular expressions needed to be adapted to: `https?://chromium-review.googlesource.com/c/REPO/+/[1-9]+[0-9]*`, where `REPO` corresponds to any of the 410 repositories within the Chromium community. The link recovery process is repeated $\forall r \in R$.

4.2 Results

By applying our link recovery approach to the six studied communities, we set out to better understand (i) the prevalence of review linkage and (ii) linkage trends over time. To do so, we measure a linkage rate for each studied system, i.e., the proportion of reviews that are connected by at least one link.

Prevalence of Linkage. Table 2 shows that 17% and 25% of Chromium and OpenStack reviews are connected by at least one link, respectively. Although the linkage rate in Qt, Eclipse, and Android reviews are lower (5%–8%), linkage is not uncommon. However, the Libreoffice linkage rate is only 3%. We suspect that this result is reflective of the differences in the importance that the studied communities have placed on code review. Indeed, reviews in the OpenStack and Chromium communities receive 17 and 12 comments on average, whereas reviews in other subject communities receive 4–8 comments on average. Two-tailed, unpaired Mann-Whitney U tests between OpenStack and the other studied communities (after Bonferroni correction to control for family-wise errors, $\alpha = \frac{0.05}{5} = 0.01$) indicate that OpenStack receives significantly more comments than the other studied communities ($p < 0.001$) with Cliff's delta effect sizes of negligible when compared to Chromium ($\delta = 0.021 < 0.147$), small when compared to Android and Qt ($0.147 \leq \delta = 0.288, 0.287 < 0.330$), medium when compared to Eclipse ($0.330 \leq \delta = 0.413 < 0.474$), and large when compared to Libreoffice ($0.474 \leq \delta = 0.580$). Furthermore, OpenStack reviews tend to involve more reviewers than the other studied communities, averaging two reviewers per review more than the next highest community (Chromium). Two-tailed, unpaired Mann-Whitney U tests indicate that OpenStack reviews have significantly more reviewers than the other studied communities ($p < 0.001$) with Cliff's delta effect sizes of medium when compared to Chromium ($0.330 \leq \delta = 0.407 < 0.474$), and large when compared to Android, Qt, Eclipse, and Libreoffice ($0.474 \leq \delta = 0.553, 0.520, 0.703, 0.712$). We also find that there is no tendency where the larger systems are likely to receive more comments in terms of an individual contribution (i.e., 2–4 comments per reviewer on average across our studied systems). Indeed, despite of the significant differences of unpaired Mann-Whitney U tests across our six studied systems, Cliff's delta effect sizes are negligible when compared to Qt and Libreoffice ($\delta = 0.051, 0.020 < 0.147$), small when compared to Android and Eclipse ($0.147 \leq \delta = 0.219, 0.213 < 0.330$), and medium when compared to Chromium ($0.330 \leq \delta = 0.357 < 0.474$).

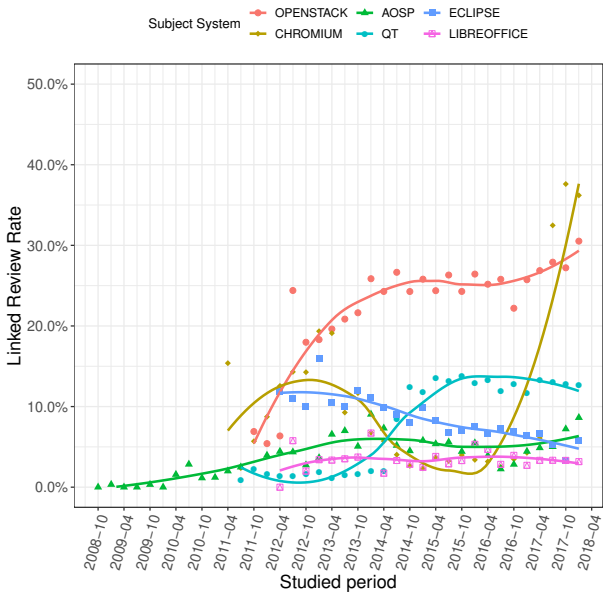

Figure 1: Monthly linkage rate in the studied communities.

Table 2 also shows that the largest proportion of links are recovered from discussion threads (*GC*, *IC*). Indeed, 72%–97% of links are recovered from *GC* and *IC* threads. This indicates that links tend to emerge during the review rather than when it is created.

Furthermore, Table 2 shows that 5%–40% of links connect reviews across project boundaries. For example, Review #102704[5] links from the Nova project to the Cinder project of OpenStack. Furthermore, OpenStack shows a greater rate of cross-project links than the other studied communities, likely because the OpenStack community develops more projects than the other studied communities. Indeed, Table 1 shows that OpenStack contains at least four times as many projects as the other studied communities.

Linkage Trends. Figure 1 shows that linkage rates in the four communities that have made the largest investments in code review (i.e., personnel and activity per review) have stabilized or peaked in the more recent studied periods. Indeed, the linkage rate in the OpenStack community has a rapidly increasing trend until late 2014 and more gradual increases in recent months. The Chromium community began with moderate linkage rates until mid-2014, when rates dropped to below 5%. However, the rates have climbed above 30%

[5] https://review.openstack.org/#/c/102704/

Table 2: Review graph characteristics in the subject communities.

Product	Linked Reviews		Per Review (Mean)		Per Reviewer	Description		General Comment		Inline Comment		Cross-project	
	#Reviews	%Reviews	Comments	Reviewers	Comments	#Links	%Links	#Links	%Links	#Links	%Links	#Links	%Links
OpenStack	133,650	25%	17	5	3	77,400	28%	151,411	54%	50,110	18%	110,964	40%
Chromium	62,065	17%	12	3	4	10,295	5%	182,929	93%	3,968	2%	44,869	23%
Android	11,584	5%	8	2	3	6,082	22%	21,491	77%	466	2%	2,508	9%
Qt	14,266	8%	8	2	3	574	3%	19,450	87%	2,309	10%	4,353	19%
Eclipse	8,227	8%	6	2	3	1,255	6%	19,995	90%	917	4%	1,359	6%
Libreoffice	1,549	3%	4	2	2	211	10%	1,774	86%	88	4%	107	5%

in mid-2017. This growth may be due to several factors (e.g., community initiative, growth in task complexity). The Qt and Android communities had linkage rates below 7% until mid-2014, when rates roughly stabilized at 13% and 9%, respectively.

On the other hand, linkage rates remain stable or are decreasing in the two studied communities that have made the least investment in code review. Indeed, the Libreoffice community shows a stable trend, while the Eclipse community's trend is decreasing.

RQ1: *Linked reviews occur regularly in communities that have made large investments in code review. Thus, code review analytics should look to linkage as a potential source of useful information.*

Figure 2: An example of a review link from Review #126831 in Nova.

5 QUALITATIVE ANALYSIS OF REVIEW LINKS (RQ2)

Reviews may be linked to other reviews for several reasons. For example, a link may express a dependency between reviews (e.g., "[Review #106918][6] is dependent on [Review #106274]") or the evolution of an earlier idea into its current form (e.g., "[Review #475649][7] is a follow up patch to [Review #411830]").

In this section, we set out to better understand the underlying reasons for review linkage through a qualitative analysis. To understand why a link has been recorded, we use a manually-intensive research method, which creates practical limitations on the breadth of communities that we can analyze. Thus, we elect to analyze the OpenStack community—the largest and most dynamic graph in our set of studied communities (see Tables 1–2 and Figure 1). The OpenStack community is composed of several projects, of which, we select the two largest for analysis, i.e., Nova (the provisioning management module) and Neutron (the networking abstraction interface). Below, we describe our approach to classify links in the studied projects (5.1) and present the results (5.2).

5.1 Approach

We apply an open coding approach [12] to classify randomly sampled links between reviews. Open coding is a qualitative data analysis method by which artifacts under inspection (review links in our case) are classified according to emergent concepts (i.e., codes). After coding, we apply open card sorting [30] to lift low-level codes to higher level concepts. Below, we describe our sampling, coding, and card sorting procedures in more detail.

Sampling. Section 4 shows that there are 133,650 linked reviews in the OpenStack community, 16,144 of which appear in the Nova and Neutron projects. Since coding of all of these links is impractical, we randomly sample Nova and Neutron review links for coding.

To discover as complete of a set of link types as possible, we strive for *saturation*. Similar to prior work [47], we set our saturation criterion to 50, i.e., we continue to code randomly selected links until no new codes have been discovered for 50 consecutive links.

To ensure that we analyze links that appear in descriptions and comments, we aim to achieve saturation twice—once when coding description-based links and again when coding comment-based links. We reach saturation after coding 340 comment-based links and 146 description-based links in Nova, and 161 comment-based links and 105 description-based links in Neutron.

Coding. Coding was performed by the first and second authors during collocated coding sessions. Both coders have experience with code reviews both in research and commercial software development settings (acting as both patch authors and reviewers). In total, these coding sessions took 56 hours (or 112 person-hours).

When coding links, the coders focused on the key reasons why the link was recorded. For example, Figure 2 shows that comments from Reviewer 1 on Nova Review #126831 have inspired the author to create Review #127457. Both coders independently code and then discuss each link until a consensus is reached. We also record the direction of the link, e.g., Review #126831 links to Review #127457.

In theory, multiple codes may apply to each link; yet in practice, we find that multi-coded links are rare. Indeed, while a link may have different codes if interpreted in different directions, we do not find any multi-coded directional links.

Since open coding is an exploratory data analysis technique, it may be prone to errors. To mitigate errors, we code in three passes. First, since codes that emerge late in the process may apply to earlier reviews, after completing an initial round of coding, we perform a second pass over all of the links to correct miscoded

[6] https://review.openstack.org/#/c/106918/
[7] https://review.openstack.org/#/c/475649/

Table 3: The frequency of the discovered types of review linkage in OpenStack Nova and Neutron.

Category	Frequency	
	Nova	Neutron
C1: Patch Dependency	269 (55%)	148 (56%)
Patch Ordering	124 (26%)	62 (24%)
Root Cause	50 (11%)	28 (11%)
Shallow Fix	6 (2%)	4 (2%)
Follow-up	28 (6%)	29 (11%)
Merge Related Reviews	19 (4%)	13 (5%)
Multi-part	42 (9%)	12 (5%)
C2: Broader Context	96 (20%)	50 (19%)
Related Feedback	43 (9%)	14 (6%)
Demonstration	29 (6%)	26 (10%)
Additional Evidence	24 (5%)	10 (4%)
C3: Alternative Solution	69 (14%)	39 (15%)
Superseding	35 (8%)	17 (7%)
Duplicated	34 (7%)	22 (9%)
C4: Version Control Issues	27 (6%)	17 (6%)
Integration Concern	15 (4%)	13 (5%)
Gerrit Misuse	5 (2%)	2 (1%)
Revert	7 (2%)	2 (1%)
C5: Feedback Related	23 (5%)	10 (4%)
Fix Related Issues	11 (3%)	3 (2%)
Feedback Inspired Reviews	12 (3%)	7 (3%)

entries. During the first coding pass, we code only using the link source (description/comment). In several cases, more contextual information was needed. We coded such cases as "Needs Additional Context" during the first coding pass. During a third coding pass, we check additional sources of information (e.g., the content of the patch, the linked review, comments in discussion threads) to code these cases more specifically. After the three coding passes, all of the sampled links have been assigned to a specific code.

Card Sorting. Similar to prior studies [3, 18, 24, 39], we apply open card sorting to construct a taxonomy of codes. This taxonomy helps us to extrapolate general themes from our detailed coded data. The card sorting process is comprised of two steps. First, the coded links are merged into cohesive groups that can be represented by a similar subgraph. Second, the related subgraphs are merged to form categories that can be summarized by a short title.

5.2 Results

Table 3 provides an overview of the categories that summarize related labels (the complete table is available online[1]). We observe that the frequencies at which the link labels appear are consistent between the two studied projects. Moreover, we only coded two of 486 links from Nova and two of 266 links from Neutron as false positives (i.e., spuriously detected links that do not indicate a relationship between reviews), suggesting that our link extraction approach does not produce much noise (precision > 0.99 in both cases). Furthermore, we required additional context information (beyond the link source) to code 63 links, all of which were more specifically coded during the third pass when we analyze additional information sources. Below, we describe the discovered codes according to the categories to which they belong.

Patch Dependency (C1). We find that 55% and 56% of the analyzed links in Nova and Neutron connect reviews to others that they depend upon. Patch Dependency links may influence integration decisions and the reviewers who should be recommended. Indeed, the integration decision in one review may be inherently linked to that of another if they share a dependency. For example, Review #102704[5] of the Nova project was only abandoned because of its dependency on Review #102705, which was abandoned earlier. Moreover, reviewers of a dependent review may need to review its dependency as well. For example, a reviewer of Review #102749[8] was added only because they reviewed its dependency (Review #101424). We further explore the usefulness of these linkage-based reviewer invitations in Section 7 (RQ4).

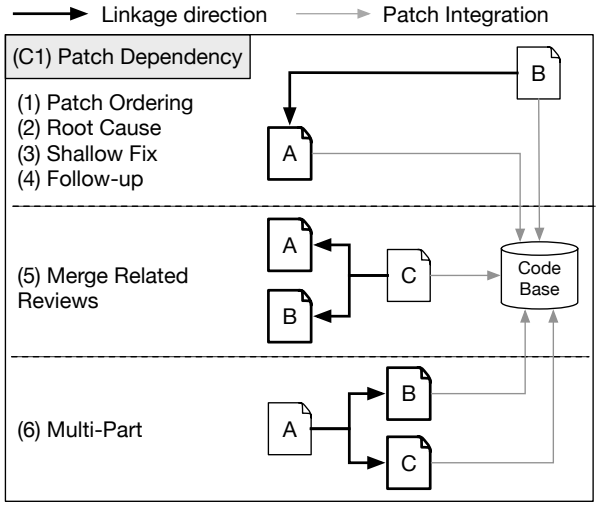

Figure 3: The Patch Dependency subgraphs.

Figure 3 shows three shapes that patch dependency links take. First, Patch Ordering, Root Cause, Shallow Fix, and Follow-up take the shape of two eventually integrated (or abandoned) reviews that share a link. While they share a shape, the semantics of the patterns differ, i.e., Patch Ordering links indicate a timing dependency that must be respected at integration time, while Follow-up, Root Cause, and Shallow Fix links provide rationale for Review B by pointing to enabling enhancements or limitations in Review A. Second, Merge Related Reviews links merge two or more reviews into a more cohesive whole. Finally, Multi-part links indicate that a large review has been split into a series of smaller reviews.

Weißgerber *et al.* [43] observed that smaller patches tend to be accepted in two large open source projects. Rigby *et al.* [35] argue that one of the statutes of an efficient and effective code review process is the "early, frequent review of small, independent, complete solutions". The frequency of the Multi-part pattern (i.e., the splitting of large patches into smaller ones) may be an indication that these prior observations still hold.

[8] https://review.openstack.org/#/c/102749/

Broader Context (C2). We find that 20% and 19% of the analyzed links point to other reviews with relevant resources. The individual analysis of reviews that are connected with Broader Context links may not be valid. Indeed, analyses of review outcome prediction often compute the length of discussion threads [16, 23]. However, a discussion may span across several reviews when Broader Context links are present. For instance, a reviewer of Review #155223[9] asks the author to refer to a similar discussion on Review #215608.

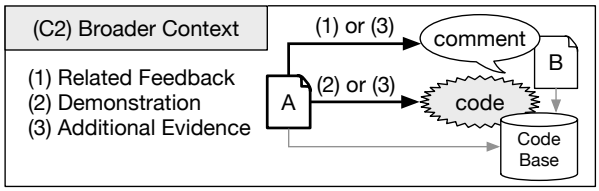

Figure 4: The Broader Context subgraph.

Figure 4 shows that our three codes within the Broader Context category share the same shape; however, the codes differ in the artifact to which they refer. Related Feedback links connect discussions on one review to discussions in other reviews, while Demonstration links point to example code from other reviews. Additional Evidence links point to other reviews as proof (code, discussions, specifications) of the existence, removal, or relevance of the problems that are addressed by the review under inspection.

Alternative Solution (C3). We find that 14% and 15% of the analyzed links connect reviews to others that implement similar functionality. Similar to Patch Dependency links, Alternative Solution links may also impact integration decisions and reviewer recommendations. For example, Review #67431[10] was abandoned because another submitted solution for the same underlying issue (Review #61041) was preferred. Especially in such examples where an "either or" decision needs to be made, the same reviewers should likely be invited to all of the competing reviews for the sake of fairness [15]. Furthermore, prior work has demonstrated that a lack of awareness of concurrently developed solutions may result in redundant work [10, 49] and is a key source of software development waste [37]. These conflated integration decisions are not congruent with review outcome or reviewer recommendation models that assume each submission is independently adjudicated [21–23].

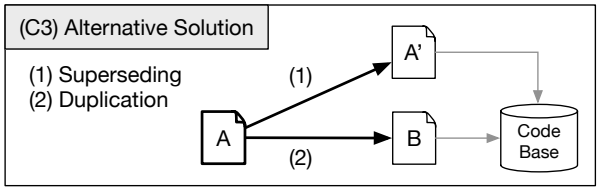

Figure 5: The Alternative Solution subgraph.

Figure 5 shows that our two codes within the Alternative Solution category share the same shape, yet differ in their semantics. Superseding links show that the solution in an earlier review has been replaced with an updated solution in the current review, while Duplication links highlight the existence of another (competing) solution to the same underlying problem. In a large-scale, cross-company software organization like OpenStack, it is difficult to coordinate development effort. However, the frequency at which work is duplicated suggests that tooling [10, 49] may help.

Version Control Issues (C4). We find that 6% of analyzed links point to reviews that introduced version control issues. Rigby and Storey [36] also found such issues are often discussed during the broadcast-based reviews in several open source systems. Shimagaki *et al.* [38] found that 5% of commits in a large industrial system were reverted after being integrated. Since Revert is one of the codes within our category, our review graphs can complement version control data to better understand the practice of reverting commits.

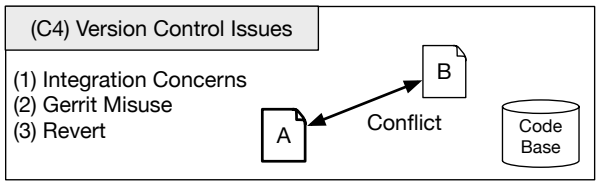

Figure 6: The Version Control Issues subgraph.

Figure 6 shows that our three codes within the Version Control Issues category share the same shape. Integration Conflict and Gerrit Misuse links expose technical integration or Gerrit issues, while Revert links indicate that a partial or complete rollback.

Feedback Related (C5). We find that 5% and 4% of analyzed links in Nova and Neutron connect reviews to others that resolve or were inspired by reviewer comments. Reviews that were inspired by feedback in another review might be more likely to be accepted, since one reviewer is already in favour of the idea. For example, in Review #167100,[11] a reviewer's feedback inspired the creation of the new Review #167082. Then, one of reviewers of #167100 joined Review #167082, and eventually approves it for integration. There are two possible ways to act upon C5-linked reviews. The reviewer who inspired the change may be well suited to review the inspired change. Thus, reviewer recommenders may need to recommend them. On the other hand, since the reviewer who inspired the change may not be impartial when reviewing the inspired review, reviewer recommenders may need to recommend other reviewers.

Figure 7 shows that our two codes within the Feedback Related category share the same shape. Fixed Related Issues links show that (part of) a raised concern has been addressed by another review. Feedback-inspired links show a new contribution where feedback on Review A inspires the creation of a new patch.

RQ2: *A broad variety of reasons for linkage exist. These different types of links may introduce noise in or opportunities for improvement of code review analytics.*

[9] https://review.openstack.org/#/c/155223/
[10] https://review.openstack.org/#/c/67431/
[11] https://review.openstack.org/#/c/167100/

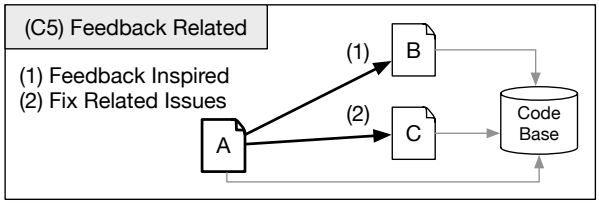

Figure 7: The Feedback Related subgraph.

6 AUTOMATED LINK CLASSIFICATION (RQ3)

In Section 5, we find that several types of links may impact reviewer recommendation and outcome prediction. Since different review analytics techniques may need to traverse or ignore links depending on the type, a more scalable approach to link type recovery is needed. Indeed, it took the authors 112 person-hours to code 752 review links (see Coding in Section 5.1). If we continue to code at this rate, it would take an additional 19,793 person-hours to code the remaining 132,898 reviews in the OpenStack data set.

In this section, we study the feasibility of using machine learning techniques to automatically classify links by categories. To do so, we use the manually coded data from Section 5 as a sample on which to train and evaluate classifiers that identify the link category (C1–C5) based on the document that it appears within (i.e., the review description field or the comment in the general or inline discussion thread). Below, we describe our approach to automated link category classification (6.1) followed by the results (6.2).

6.1 Classification Approach

Feature Extraction. We apply standard text preprocessing techniques to lessen the impact of noise on our classifiers. We first tokenize the document and remove stop words using the Python NLTK stop word list. Next, we apply lemmatization to handle term conjugation using the Python NLTK `lemmatize` function. Finally, we convert each sampled description or comment to a vector of the Term Frequency-Inverse Document Frequency (TF-IDF) weights of its terms. Broadly speaking, terms that appear rarely across documents, and/or often within one document are of higher weight. We use the Python Scikit-Learn `TfidfVectorizer` function to compute TF-IDF scores for all documents in a training sample.

Classifier Validation Technique. To estimate classifier performance on unseen data, we apply the out-of-sample bootstrap validation technique [13], which tends to yield more robust results than other validation techniques (e.g., k-fold cross validation) [40]. First, a bootstrap sample of size N is randomly drawn with replacement from the original sample of the same size N. This bootstrap sample is used to train our classifiers, while the documents from the original sample that do not appear in the bootstrap sample are set aside for testing. Since the bootstrap sample is selected with replacement, on average, 36.8% of the documents will not appear in the bootstrap sample and can be used to evaluate classifier performance. We perform 1,000 iterations of the bootstrap procedure (reporting the mean performance scores across these iterations) to ensure that our performance measurements are robust.

Classification Techniques. To train our classifiers, we experiment with a broad selection of popular classification techniques. Support Vector Machines (SVM) use a hyperplane to classify documents by first transforming feature values into a multidimensional feature space. Random Forest is an ensemble learning technique that builds a large number of decision trees, each using a subset of the features, and then aggregates the results from each tree to classify documents. Multinomial Naïve Bayes (MNB) is a conditional probability model that uses a multinomial distribution for each of the features. Multi-Layer Perceptron (MLP) is a supervised learning technique where weighted inputs are delivered through neurons in sequential layers. Multinomial Logistic Regression (MLR) generalizes the logistic regression technique to the multi-class classification setting. We use the Python Scikit-Learn implementations of the classification techniques (`svm.SVC`, `RandomForestClassifier`, `MultinomialNB`, `MLPClassifier`, and `LogisticRegression`).

Hyperparameter Optimization. The classification techniques that we use have configurable parameters that impact their performance. Similar to prior work [40], we use a grid search to tune the parameter settings. Grid search is an exhaustive searching technique that examines all of the combinations of a specified set of candidate settings to find the best combination. We explore the same set of candidate settings as Tantithamthavorn *et al.* [40, p. 5]. We search for the optimal parameter settings for each classification technique in each bootstrap sample (i.e., without using the testing data) using the Scikit-Learn `GridSearchCV` function.

Performance Evaluation. To evaluate our classifiers, we use common performance measures. Precision is the proportion of links that are classified as a given category that are correct. Recall is the proportion of links of a given category that a classifier can detect. The F1-score is the harmonic mean of precision and recall. The Area Under the Curve (AUC) computes the area under the curve that plots the true positive rate against the false positive rate as the threshold that is used for classifying documents varies. AUC ranges from 0 to 1, where random guessing achieves an AUC of 0.5.

Since our links have more than two categories, we need to use multi-class generalizations of these performance measures. Each measure is computed for each category before being aggregated into an overall score. Since the link categories are imbalanced (see Table 3), we weigh the category scores by their overall proportion.

We also compare our classifiers to a ZeroR classifier, which always reports the most frequently occurring class. In our setting, a ZeroR classifier achieves a recall of one and a precision equal to the frequency of the most frequently occurring category (C1) for that class, and a precision and recall of zero for the other categories. Note that AUC does not apply to ZeroR classifiers because likelihood estimates are not produced. We use the Scikit-Learn `metrics` library to compute our performance measurements.

6.2 Results

Table 4 shows that while no classification technique consistently outperforms the others, the classifiers achieve a precision of 0.71–0.77, a recall of 0.72–0.92, and F1-scores of 0.71–0.79. Since these performance scores are on par with those of prior classification studies [28, 32, 33], we believe that our classifiers show promise. Moreover, Table 4 shows that our classifiers outperform baseline

Table 4: The performance of our five link category classifiers, all of which outperform the ZeroR and random guessing baselines in all cases.

Model	Nova				Neutron			
	Prec.	Rec.	F1.	AUC	Prec.	Rec.	F1.	AUC
SVM	0.75	0.88	0.79	0.72	0.72	0.92	0.77	0.82
RF	0.77	0.72	0.71	0.57	0.76	0.84	0.72	0.89
MNB	0.71	0.80	0.74	0.72	0.71	0.82	0.75	0.76
MLP	0.74	0.81	0.76	0.61	0.75	0.84	0.74	0.68
MLR	0.75	0.88	0.79	0.65	0.72	0.92	0.77	0.79
ZeroR	0.26	0.50	0.34	–	0.27	0.50	0.34	–

approaches, achieving precision, recall, and F1-scores that are 22–51 percentage points better than the ZeroR baseline and AUC values above the 0.5 random guessing benchmark.

RQ3: *Despite the complexity of the five-class classification problem, our classifiers achieve precision, recall, and F1-scores that exceed the ZeroR baseline by 22–51 percentage points.*

7 LINKAGE IMPACT ANALYSIS (RQ4)

Our analysis in RQ2 suggests that linkage can impact code review analytics. In this section, we measure that impact on reviewer recommendation (7.1) and review outcome prediction (7.2).

7.1 Reviewer Recommendation

Approach. In Section 5, we observe that Patch Dependency links (C1), Alternative Solution links (C3), and Feedback Related links (C5) may impact reviewer recommendation because reviewers of a linking review may need to review its linked review.

To measure the degree to which reviewers contribute to both linking and linked reviews, we compute the *Overlapping Reviewer Rate* (ORR), i.e., the proportion of reviewers from the linking review who also review the linked review. We compute the ORR rates on our sampled links from Section 5.

To study the extent to which state-of-the-art reviewer recommenders identify overlapping reviewers, we apply cHRev [48], which makes recommendations based on prior contribution and working habits. We approximate data sets where C1, C3, and C5-linked reviews are known by applying our top-performing classifier (SVM) from Section 6 to the linked Nova and Neutron reviews. We exclude C5-linked reviews from further analysis because we detect too few instances (i.e., five) to draw meaningful conclusions.

We then compare the performance of cHRev to an extended version that ranks reviewers of linking reviews at the top of the list for linked reviews. Since links appear as reviews evolve, we select only those links that are available at prediction times zero, one, three, and six hours after the review has been created. Moreover, we only identify candidate overlapping reviewers who have commented on linking reviews at or before those prediction time settings.

Results. The ORR rates of C1 and C3 are 50%–51% and 65%–77%, respectively. By way of comparison, we find that the ORR rate is less than 1% for randomly selected pairs of non-linked reviews. The results indicate that reviewers are more likely to participate in both reviews when links are present.

A closer inspection reveals that cHRev misses at least one overlapping reviewer in 96%–100% of linked reviews across Top 1–5 recommendation lists. Since medians of 12 and 15 reviewers participate in Nova and Neutronreviews, respectively, missing at least one overlapping reviewer is a concern. Moreover, none of the overlapping reviewers are recommended in 41%–81% of Nova and 48%–84% of Neutron reviews. When overlapping reviewers are omitted, they appear in 13th–15th place on average. Increasing the weight of overlapping reviewers may improve recommendations.

Table 5 shows that the precision, recall, and F1-scores of cHRev improve by 33%–88% (3–17 percentage points) if reviewers of a linked review are recommended at the top of the list. Moreover, the degree of the improvement remains consistent across the four studied prediction time settings, indicating that no prediction delay is necessary to gain the bulk of the value. Longer delays may achieve better results but would be less useful in practice, since waiting more than six hours for reviewer recommendations is impractical.

7.2 Review Outcome

Approach. In Section 5, we report that C1, C2 (Broader Context), and C3 links may impact review outcome prediction because the integration decision in one review may be inherently linked to that of the other. Similar to the reviewer recommendation experiment above, we apply our SVM classifier to identify C1, C2, and C3 links in the full Nova and Neutron data sets.

For C1, C2, and C3 reviews, we compute the *Identical Outcome Rate* (IOR), i.e., the rate at which linking and linked reviews result in the same outcome (i.e., integration or abandonment). Moreover, to study the extent to which state-of-the-art outcome predictors misclassify identical outcomes, we apply the outcome prediction approach of Gousios *et al.* [16]. More specifically, we train prediction models that classify reviews as integrated or abandoned based on review properties (e.g., # comments, # participants). In our setting, we train the predictors on unlabeled linked reviews and evaluate them on our labeled samples.

Results. The IOR rates of integrated C1 and C2-linked reviews are 73%–87% in Nova and 55%–71% in Neutron, while the IOR rates for abandonment are 57%–86% and 45%–75%, respectively. This suggests that reviews that are connected with C1 and C2 links tend to have the same outcome. Since C3 links connect competing solutions, it is unlikely that they will have the same outcome. This is reflected in lower IOR rates of 18%–26% for integration, respectively. On the other hand, the IOR rates for abandonment are 46% in Nova and 62% in Neutron, indicating that it is not uncommon for both competing solutions to be abandoned.

Outcome predictors may misclassify review outcomes for linked reviews when the feature values span multiple reviews (e.g., discussion contexts, # participants). Indeed, we find that prior approach misclassifies 35% and 39% of C1, C2, and C3-linked reviews in Nova and Neutron, respectively.

RQ4: *Reviewer recommenders tend to omit or poorly rank reviewers who participate in both linking and linked reviews. Moreover, review outcome predictors tend to misclassify linked review outcomes. Leveraging links that are available at prediction time can yield considerable performance improvements.*

Table 5: The mean performance scores of cHRev [48] and our proposed link-aware reviewer recommenders at different prediction delays. The bulk of the performance improvement is achieved in the no delay (0 hour) setting.

	Nova									Neutron								
	Top 1			Top 3			Top 5			Top 1			Top 3			Top 5		
Model	Pre.	Rec.	F1.	Pre.	Rec.	F1.	Pre.	Rec.	F1.	Pre.	Rec.	F1.	Pre.	Rec.	F1.	Pre.	Rec.	F1.
Baseline	0.25	0.06	0.09	0.28	0.18	0.22	0.30	0.32	0.30	0.24	0.05	0.08	0.26	0.17	0.20	0.28	0.31	0.29
0 hour	0.39	0.09	0.14	0.40	0.26	0.31	0.40	0.43	0.41	0.39	0.09	0.14	0.40	0.26	0.31	0.41	0.44	0.42
1 hour	0.39	0.09	0.14	0.40	0.26	0.31	0.40	0.43	0.41	0.41	0.09	0.14	0.41	0.26	0.32	0.42	0.44	0.43
3 hours	0.40	0.09	0.14	0.40	0.26	0.31	0.40	0.43	0.41	0.41	0.09	0.14	0.41	0.26	0.32	0.42	0.45	0.43
6 hours	0.40	0.09	0.14	0.40	0.26	0.31	0.40	0.43	0.41	0.41	0.09	0.15	0.41	0.27	0.32	0.42	0.45	0.43

8 PRACTICAL IMPLICATIONS

Linkage should be taken into consideration in code review analytics. Our quantitative study (Section 4) shows that up to 25% of reviews in our subject communities link to at least one other review. Moreover, in the four studied communities that have made the largest investment in reviewing (OpenStack, Chromium, Android, and Qt), we observe increasing or stable trends in the monthly linkage rate. Prior work has shown that linked artifacts can impact repository mining analytics [8, 20]. Indeed, our impact analysis (Section 7) shows that reviewer recommenders and outcome predictors can be improved by taking linkage into account.

Duplicate detection would enhance review efficiency. Our qualitative study shows that 14%–15% of congruent links in Nova and Neutron fall in the Alternative Solution category (i.e., superseding and duplication). Duplicate contributions are a form of waste in software development [37]. Boisselle and Adams [8] argued that automatic classification of bug reports that are linked due to duplication would be helpful. However, in large projects like OpenStack and Chromium, it is difficult to keep track of duplicated work [49]. Systematic detection of duplicates would be important from a review fairness [15] perspective as well, since the competing solutions should be subject to the same level of scrutiny.

Link category recovery may not be necessary. The majority of links are of types that potentially impact reviewer recommendation (C1, C3, and C5) and outcome prediction (C1, C2, and C3). Since we find that all studied link categories are useful, future work may assume that all links are useful and omit link type classification.

9 THREATS TO VALIDITY

Construct validity. Construct validity is concerned with the degree to which our measurements capture what we aim to study. We recover links between reviews using regular expressions that detect Change IDs and URLs of other reviews. However, these regular expressions may extract Change IDs or URLs that are not intended to be links (false positives). On the other hand, Section 5 shows that the false positive rate in our manually analyzed sample is quite low (<0.1%). Thus, we false positives do not appear to be of concern.

In our qualitative analysis, links may be miscoded due to the subjective nature of our open coding approach. We take several precautions to mitigate the miscoding threat. First, the code for each link is agreed upon by two coders who have experience with code review in academic and commercial settings. Furthermore, we employ a three-pass approach, where each code is revisited at least once to ensure that the correct code was selected.

Internal validity. Internal validity is concerned with our ability to draw conclusions from the relationship between study variables. Links may not be detected in all of the cases when they should be, which may introduce noise in our linkage rate observations in Section 4. Hence, our observations in Section 4 should be interpreted as lower bounds rather than as exact linkage rate values.

If we stop coding too early during our qualitative analysis (Section 5), it may threaten the completeness of the discovered set of link types. To mitigate the threat, we continue to code until our samples saturate—a concept that we operationalized by continuing until we coded a span of 50 coded links without discovering any new codes. Others have used similar saturation criteria [36, 47].

Other classification techniques or hyperparameter settings may yield better results than the ones that we studied in Section 6. To combat this, we select a broad set of popular classification techniques and use an automatic parameter optimization approach to select the best configuration of hyperparameter settings for each bootstrap iteration. Nonetheless, exploration of other classification techniques and hyperparameter settings may yield better results.

External validity. External validity is concerned with our ability to generalize based on our results. We extract review graphs (Section 4) from six communities that use the Gerrit code review tool. Due to the manually intensive nature of our coding approach, we focus on an in-depth analysis of the two largest projects from the OpenStack community. As such, our linkage types, classifiers, and impact analyses may not generalize to code reviewing environments in all software communities. Replication studies may be needed to arrive at more general conclusions. To simplify replication, we have made our scripts and data publicly available.[1]

10 CONCLUSION

Researchers have recently proposed several analytics-based techniques to support stakeholders in the MCR process. However, those techniques have tacitly or explicitly treated each review as an independent observation, which overlooks relationships among reviews.

Our empirical study suggests that linkage is not uncommon in six studied software communities. Moreover, adding linkage awareness to review analytics approaches yields considerable performance improvements. Thus, review linkage should be taken into consideration in future MCR studies and tools.

ACKNOWLEDGMENTS

This work was supported by the SCAT Technology Research Foundation and JSPS KAKENHI Grants JP17J09333, 17H00731, 18KT0013.

REFERENCES

[1] Sultan S. Alqahtani, Ellis E. Eghan, and Juergen Rilling. 2017. Recovering Semantic Traceability Links between APIs and Security Vulnerabilities: An Ontological Modeling Approach. In *Proceedings of International Conference on Software Testing, Verification and Validation*. 80–91.

[2] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. 2002. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering* 28, 10, 970–983.

[3] Alberto Bacchelli and Christian Bird. 2013. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proceedings of the 35th International Conference on Software Engineering*. 712–721.

[4] Vipin Balachandran. 2013. Reducing Human Effort and Improving Quality in Peer Code Reviews Using Automatic Static Analysis and Reviewer Recommendation. In *Proceedings of the 35th International Conference on Software Engineering*. 931–940.

[5] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. 2012. The Secret Life of Patches: A Firefox Case Study. In *Proceedings of the 19th Working Conference on Reverse Engineering*. 447–455.

[6] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. 2013. The influence of non-technical factors on code review. In *Proceedings of the 20th Working Conference on Reverse Engineering*. 122–131.

[7] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey. 2016. Investigating Technical and Non-Technical Factors Influencing Modern Code Review, In Empirical Software Engineering. *Empirical Software Engineering* 21, 3, 932–959.

[8] Vincent Boisselle and Bram Adams. 2015. The impact of cross-distribution bug duplicates, empirical study on Debian and Ubuntu. In *Proceedings of the 15th International Working Conference on Source Code Analysis and Manipulation*. 131–140.

[9] Amiangshu Bosu, Michaela Greiler, and Christian Bird. 2015. Characteristics of Useful Code Reviews: An Empirical Study at Microsoft. In *Proceedings of the 12th International Conference on Mining Software Repositories*. 146–156.

[10] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2011. Proactive Detection of Collaboration Conflicts. In *Proceedings of the 8th Joint Meeting on European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*. 168–178.

[11] Gerardo Canfora, Luigi Cerulo, Marta Cimitile, and Massimiliano Di Penta. 2011. Social Interactions Around Cross-system Bug Fixings: The Case of FreeBSD and OpenBSD. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. 143–152.

[12] Kathy Charmaz. 2014. *Constructing Grounded Theory*. SAGE Publications.

[13] Bradley Efron and Robert J. Tibshirani. 1993. An Introduction to the Bootstrap. *Chapman & Hall*.

[14] Michael E. Fagan. 1976. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal* 15, 3, 182–211.

[15] Daniel M. German, Gregorio Robles, Germán Poo-Caamaño, Xin Yang, Hajimu Iida, and Katsuro Inoue. 2018. "Was My Contribution Fairly Reviewed?": A Framework to Study the Perception of Fairness in Modern Code Reviews. In *Proceedings of the 40th International Conference on Software Engineering*. 523–534.

[16] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An Exploratory Study of the Pull-based Software Development Model. In *Proceedings of the 36th International Conference on Software Engineering*. 345–355.

[17] Jin L. C. Guo, Jinghui Cheng, and Jane Cleland-Huang. 2017. Semantically Enhanced Software Traceability Using Deep Learning Techniques. In *Proceedings of the 39th International Conference on Software Engineering*. 3–14.

[18] Anja Guzzi, Alberto Bacchelli, Michele Lanza, Martin Pinzger, and Arie van Deursen. 2013. Communication in Open Source Software Development Mailing Lists. In *Proceedings of the 10th International Conference on Mining Software Repositories*. 277–286.

[19] Kazuki Hamasaki, Raula Gaikovina Kula, Norihiro Yoshida, A. E. Camargo Cruz, Kenji Fujiwara, and Hajimu Iida. 2013. Who Does What During a Code Review? Datasets of OSS Peer Review Repositories. In *Proceedings of the 10th International Conference on Mining Software Repositories*. 49–52.

[20] Hideaki Hata, Christoph Treude, Raula G. Kula, and Takashi Ishio. 2019. 9.6 Million Links in Source Code Comments: Purpose, Evolution, and Decay. In *Proceedings of the 41st International Conference on Software Engineering*. 1211–1221.

[21] Vincent J. Hellendoorn, Premkumar T. Devanbu, and Alberto Bacchelli. 2015. Will They Like This? Evaluating Code Contributions with Language Models. In *Proceedings of the 12th International Conference on Mining Software Repositories*. 157–167.

[22] Gaeul Jeong, Sunghun Kim, Thomas Zimmermann, and Kwangkeun Yi. 2009. *Improving Code Review by Predicting Reviewers and Acceptance of Patches*. Technical Report. Reserach On Software Analysis for Error-free Computing. 215–226 pages.

[23] Yujuan Jiang, Bram Adams, and Daniel M. German. 2013. Will My Patch Make It? And How Fast?: Case Study on the Linux Kernel. In *Proceedings of the 10th International Conference on Mining Software Repositories*. 101–110.

[24] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. 2014. Why do Automated Builds Break? An Empirical Study. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution*. 41–50.

[25] Oleksii Kononenko, Olga Baysal, and Michael W. Godfrey. 2016. Code Review Quality: How Developers See It. In *Proceedings of the 38th International Conference on Software Engineering*. 1028–1038.

[26] Vladimir Kovalenko, Nava Tintarev, Evgeny Pasynkov, Christian Bird, and Alberto Bacchelli. 2018. Does Reviewer Recommendation Help Developers? *IEEE Transactions on Software Engineering*.

[27] Lisha Li, Zhilei Ren, Xiaochen Li, Weiqin Zou, and He Jiang. 2018. How are Issue Units Linked? Empirical Study on the Linking Behavior in GitHub. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference*. 386–395.

[28] Zhixing Li, Yue Yu, Gang Yin, Tao Wang, Qiang Fan, and Huaimin Wang. 2017. Automatic Classification of Review Comments in Pull-based Development Model. In *Proceedings of the 29th International Conference on Software Engineering and Knowledge Engineering*.

[29] Wanwangying Ma, Lin Chen, Xiangyu Zhang, Yuming Zhou, and Baowen Xu. 2017. How Do Developers Fix Cross-Project Correlated Bugs? A Case Study on the GitHub Scientific Python Ecosystem. In *Proceedings of the 39th International Conference on Software Engineering*. 381–392.

[30] Peter Morville and Louis Rosenfeld. 2006. *Information Architecture for the World Wide Web: Designing Large-Scale Web Sites*. O'Reilly Media.

[31] Mohammad M. Rahman, Chanchal K. Roy, and Jason A. Collins. 2016. CORRECT: Code Reviewer Recommendation in GitHub Based on Cross-Project and Technology Experience. In *Proceedings of the 38th International Conference on Software Engineering*. 222–231.

[32] Mohammad Masudur Rahman, Chanchal K. Roy, and Raula G. Kula. 2017. Impact of Continuous Integration on Code Reviews. In *Proceedings of the 14th International Conference on Mining Software Repositories*. 499–502.

[33] Mohammad Masudur Rahman, Chanchal K. Roy, and Raula G. Kula. 2017. Predicting Usefulness of Code Review Comments Using Textual Features and Developer Experience. In *Proceedings of the 14th International Conference on Mining Software Repositories*. 215–226.

[34] Michael Rath, Jacob Rendall, Jin L. C. Guo, Jane Cleland-Huang, and Patrick Mäder. 2018. Traceability in the Wild: Automatically Augmenting Incomplete Trace Links. In *Proceedings of the 40th International Conference on Software Engineering*. 834–845.

[35] Peter C. Rigby, Daniel M. German, and Margaret-Anne 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*. 541–550.

[36] Peter C. Rigby and Margaret-Anne Storey. 2011. Understanding Broadcast Based Peer Review on Open Source Software Projects. In *Proceedings of the 33rd International Conference on Software Engineering*. 541–550.

[37] Todd Sedano, Paul Ralph, and Cécile Péraire. 2017. Software Development Waste. In *Proceedings of the 39th International Conference on Software Engineering*. 130–140.

[38] Junji Shimagaki, Yasutaka Kamei, Shane McIntosh, David Pursehouse, and Naoyasu Ubayashi. 2016. Why are Commits being Reverted? A Comparative Study of Industrial and Open Source Projects. In *Proceedings of the 32nd International Conference on Software Maintenance and Evolution*. 301–311.

[39] Mini Shridhar, Bram Adams, and Foutse Khomh. 2014. A Qualitative Analysis of Software Build System Changes and Build Ownership Styles. In *Proceedings of the 8th International Symposium on Empirical Software Engineering and Measurement*. 29:1–29:10.

[40] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. 2018. The Impact of Automated Parameter Optimization on Defect Prediction Models. *IEEE Transactions on Software Engineering*, 1–1.

[41] Yida Tao, Donggyun Han, and Sunghun Kim. 2014. Writing Acceptable Patches: An Empirical Study of Open Source Project Patches. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution*. 271–280.

[42] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Kenichi Matsumoto. 2015. Who Should Review My Code? A File Location-Based Code-Reviewer Recommendation Approach for Modern Code Review. In *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering*. 141–150.

[43] Peter Weißgerber, Daniel Neu, and Stephan Diehl. 2008. Small patches get in!. In *Proceedings of the 5th International Conference on Mining Software Repositories*. 67–76.

[44] Xin Xia, David Lo, Xinyu Wang, and Xiaohu Yang. 2015. Who Should Review This Change? Putting Text and File Location Analyses Together for More Accurate Recommendations.. In *Proceedings of the 31st International Conference on Software Maintenance and Evolution*. 261–270.

[45] Xin Yang, Raula Gaikovina Kula, Norihiro Yoshida, and Hajimu Iida. 2016. Mining the Modern Code Review Repositories: A Dataset of People, Process and Product. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 460–463.

[46] Fiorella Zampetti, Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, and Michele Lanza. 2017. How Developers Document Pull Requests with External References. In *Proceedings of the 25th International Conference on Program Comprehension*. 23–33.

[47] Farida El Zanaty, Toshiki Hirao, Shane McIntosh, Akinori Ihara, and Kenichi Matsumoto. 2018. An Empirical Study of Design Discussions in Code Review. In *Proceedings of the 12th International Symposium on Empirical Software Engineering and Measurement*. 11:1–11:10.

[48] Motahareh Zanjani, Huzefa Kagdi, and Christian Bird. 2015. Automatically Recommending Peer Reviewers in Modern Code Review. *IEEE Transactions on Software Engineering* 42, 6, 530–543.

[49] Shurui Zhou, Ştefan Stănciulescu, Olaf Leßenich, Yingfei Xiong, Andrzej Wąsowski, and Christian Kästner. 2018. Identifying Features in Forks. In *Proceedings of the 40th International Conference on Software Engineering*. 105–116.