2016 IEEE/ACM 13th Working Conference on Mining Software Repositories

Mining the Modern Code Review Repositories: A Dataset of People, Process and Product

Xin Yang*, Raula Gaikovina Kula†, Norihiro Yoshida‡, and Hajimu Iida*

*NAIST, Japan
kin-y@is.naist.jp, iida@itc.naist.jp

†Osaka University, Japan
raula-k@ist.osaka-u.ac.jp

‡Nagoya University, Japan
yoshida@ertl.jp

ABSTRACT

In this paper, we present a collection of Modern Code Review data for five open source projects. The data showcases mined data from both an integrated peer review system and source code repositories. We present an easy–to–use and richer data structure to retrieve the (a) People, (b) Process, and (c) Product aspects of the peer review. This paper presents the extraction methodology, the dataset structure, and a collection of database dumps[1].

1. INTRODUCTION

In recent years, much research has been done on mining the modern code review repositories [6, 11, 18]. Code review is regarded as an important research field in software engineering [7]. Previously, various aspects of code review have been studied by conducting controlled and empirical experiments [13, 15, 20].

Nowadays, many OSS use a modern code review system (e.g., Gerrit, Rietveld) to archive the records of code review activities in their repositories. Many researchers in the MSR field have used these archives for the empirical investigation of code review [9, 11, 14, 18]. Each research group developed their own individual datasets for mining code repositories. However, we need a dataset that can be replicated and used as a benchmark to test related techniques and tools.

As a result, Mukadam et al. [12] and our research group [8] published datasets in the 2013 MSR data showcase. However, compared to our older dataset, the presented dataset is comprised of more projects and has a richer set of content for researchers. Based on the official REST API, our dataset extracts only the key data attributes needed to reconstruct specific aspects of the peer review process.

We present our data in an easy–to–use relational database, thus making it easy for researchers to import into their tools and techniques. Concretely, we show how the dataset can be utilized to study code review from three aspects; (i) people (ii) process and (iii) product-related aspects of code review.

[1]http://kin-y.github.io/miningReviewRepo (Feb 18, 2016)

MSR'16, May 14-15, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2901739.2903504

2. PEER REVIEW CONCEPTS

We designed our dataset by identifying these three essential aspects that are related to the code review research, as shown in Figure 1.

1. `People-related:` refers to social features of software development teams, reviewer roles, and types. Leveraging the socio-technical aspects, we investigate teamwork and collaboration of code members. This can be beneficial to the quality and efficiency of the end product. Typical topics of interest that could be mined are knowledge sharing, collaboration and information flows, code component ownership, and hierarchy within the software team.

2. `Process-related:` refers to review process and review states that are involved in the modern code peer review. Effective and efficient software processes allow for better quality of the code review, which results in a higher quality product. Mining these processes can be utilized to reduce the review time, while making assignments of skilled reviewers to every review.

3. `Product-related:` entails code change, the reviewed code patchset, and associated files. Finally, studying the submitted and merged code patches provides insights into quality aspects, answering such research questions like *'what is the ideal patch size?'* to *'what are critical elements of a successful or unsuccessful patch?'*. Program analysis techniques and code metrics can be utilized to this end.

We now discuss the peer review terms used in this paper. **People–People Types**. In a code review, we distinguish the different roles assigned to members of the review community. An `author/submitter` represents the developer who submits a change to Gerrit, and is the owner of this change. A `committer` represents the contributor who has the authority to commit the change to the source code repository. A `reviewer` represents the contributor who performs the code review to any submitted code change. A `verifier` is responsible for building, testing and verifying the changes and decides whether it is suitable for merging. `Verifiers` could be either human or automatic tools (e.g., OpenStack runs testing scripts in Jenkins CI as verifiers). An `approver` is an experienced reviewer who has the authority to approve the changes. An `approver` approves any changes by checking whether the changes fit the best practices established by

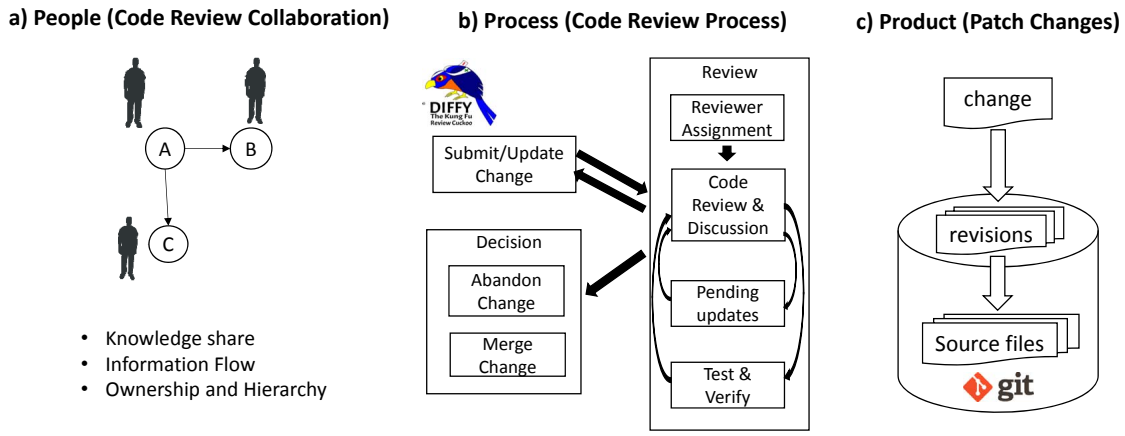

Figure 1: Depicts our proposed aspects of mining a.) People, b.) Process and c.) Product aspects of code review.

Table 1: Dataset Statistics. Projects refers to the number of source code repositories per target.

Project	Time	# Patches	# Reviewers	# Reviews	# Projects	DB Size
OpenStack	2011/07∼2015/04	173,749	5,091	3,961,771	611	1.95 GB
LibreOffice	2012/03∼2015/05	13,597	437	66,618	20	56.5 MB
AOSP	2008/10∼2015/04	63,610	3,334	355,765	567	279 MB
Qt	2011/05∼2015/04	110,172	1,437	1,062,105	111	1.41 GB
Eclipse	2012/02∼2015/05	9,168	759	84,953	189	61.9 MB

the project; assessing whether the changes fits the project's stated purpose and the existing architecture. Some projects refer to `approvers` as `core reviewers`. As shown in Figure 1 a.), we can use the reviewer types to create a social network, which is useful to analyze social interactions such as knowledge sharing or information flows within the review community.

Process–Code Review States. Every project usually follows a customized workflow, such as the AOSP project[2]. However, as shown in Figure 1 b.), most projects follow these three generic states of code review: `open`, `merged`, and `abandoned` states. An `open` change indicates that a change has not been merged into the source code repository. A `merged` change indicates that the change has already been merged into source code repository, while an `abandoned` change indicates that the change cannot be merged for certain reasons.

As shown in Figure 1 b.), the code review states indicates different stages in the code review process. The `open` state can be divided into `new`, `merge conflict` and many other states, specific to a projects workflow. Every change must start from `new` state once the author has submitted it. `merged` and `abandoned` can be regarded as the final decisions of a `open` change. The final decisions of changes usually come from the code review, testing and discussions of core reviewers, which have high authorities in Gerrit system. In addition, projects can tailor their code review states to meet their own needs (e.g., Qt have specialized review states: `Staged`, `Integrating` and `Deferred`).

Product–Code Changes. Shown in Figure 1 c.) the code review includes the code changes related to a code review. When a author commits source code, Gerrit will generate a unique `change-id` and create a new `change` in server if not exists. When the author commits a new version of the `change`, it is regarded as a `revision` in Git (it also can be called as a Patch Set in Gerrit). Through Gerrit web server, reviewers can observe the lists of the complete file paths of related `files` in each `revision`, and the summaries of source code changes to files (number of inserted lines and deleted lines). Furthermore, the specific source code changes of files can be observed by showing the *diff* of two different revisions.

3. MINING METHODOLOGY

Extraction rationale Our dataset is an extraction of the Gerrit repositories through Gerrit official REST API[3]. Using the REST API, we obtained a raw Gerrit dataset from Gerrit servers by sending API requests. The received response will be in the form of a JSON format

However, we identified two reasons why researchers may find it difficult to use the JSON format:

- Complex querying - Querying the JSON format for aspects such as the reviewer types or the process states can quickly become tedious.
- Portability - We would like to represent the data in a format that is easily imported into researchers analysis tools. Thus, we transform the data into a relational database format.

Mining Scripts. We have created a set of mining scripts, which allowed us to mine the dataset easily. Specifically, we choose Python to develop the mining scripts, with MySQL to store the extracted dataset. In Section 4, we introduce the detail of review dataset and the database schema.

[2]https://source.android.com/source/life-of-a-patch.html (Feb 18, 2016)

[3]https://gerrit-documentation.storage.googleapis.com/Documentation/2.11.1/rest-api.html (Feb 18, 2016)

Table 2: Relational database schema with attributes

Table	Key	Attribute(Definition)
`Change` (t_change)	PK	**id**(Unique change id (auto increment))
		ch_Id(Combination of project name, branch name and change id)
		ch_changeId(Change id in Gerrit)
		ch_project(Project name of change)
		ch_branch(Branch name of change)
	FK	**ch_authorId**(Author of change)
		ch_createdTime(Timestamp of when change was created)
		ch_status(Review status of change)
`Revision` (t_revision)	PK	**id**(Unique revision id (auto increment))
		rev_Id(Commit id of revision)
		rev_subject(Subject of revision)
		rev_message(Message of revision)
		rev_authorName(Author of the revision)
		rev_createdTime(Timestamp of when revision was created)
		rev_committerName(Committer of revision)
		rev_committedTime(Timestamp of when revision was committed)
	PK	**rev_patchSetNum**(Revision number in change)
	FK	**rev_changeId**(Change that the revision belongs to)
`People` (t_people)	PK	**id**(Unique people id (auto increment))
		p_authorId(Id of author)
		p_authorName(Name of author)
		p_email(Email address of author)
		p_domain(Domain of email address)
`History` (t_history)	PK	**id**(Unique comment id (auto increment))
		hist_id(Comment id in UUID form)
		hist_message(Comment message)
	FK	**hist_authorId**(Author of comment)
		hist_createdTime(Timestamp of when comment was created)
	FK	**hist_patchSetNum**(Revision number that comment was created for)
	FK	**hist_changeId**(Change that comment was created for)
`File` (t_file)	PK	**id**(Unique file ID (auto increment))
		f_fileName(The path and name of file)
		f_linesInserted(# of inserted lines)
		f_linesDeleted(# of deleted lines)
	FK	**f_revisionId**(Revision that file belongs to)

To obtain the changed code from the pending Git repositories, the following script can be used: `git ls-remote | grep [change-id]`. This will list all the commit-id and path of revisions for a change. The Git command then can be used to obtain certain revision and the diff. Similar useful scripts[4] were used for other aspects of extraction.

Challenges and Limitations. The main challenge faced when mining the repositories, was the adaptation of the mining script to correctly extract the data from each project. This is because each project has customized the review process. As a result, we had to modify our mining scripts to fit the different API versions of each Gerrit server. For example, we had to change our scripts for the AOSP project, as it adopted the newer version of Gerrit API.

Since reviewer profiles are based on the registration, a possible threat is email aliasing. This is where members may use multiple accounts. We propose for future work to use a semi-manual process of cross-checking the username, name, and email address to remove duplicates. In addition, we currently only identify the file path and size of the patch submitted. For future work, we would like to capture the actual source code changed.

[4]https://github.com/saper/gerrit-fetch-all (Feb 18, 2016)

4. DATASET

Mined Repositories. The core of the dataset comes from projects that use GIT as their source code repository, and are also integrated with the GERRIT[5] modern code review system. As shown in Table 1, the datasets comprise of five large-scale open source projects. All projects are hosted online and are accessible through their respective web interfaces [1, 2, 3, 4, 5]. The largest project is OpenStack, a cloud operation system. It has over 3,900,000 reviews and just over 5,000 reviewers. The smallest project collected is LibreOffice, with just over 66,600 reviews and 437 reviewers. We compressed the datasets files to RAR and 7z format for each project. All files are available online for download.

Dataset Schema. We transformed the JSON format into our database schema. Each attribute of the tables can be found in Table 2. Our data structure is consistent with the official Gerrit REST API. A full description of the database schema is available[6]. We summarize the descriptions of the five tables below:

- **Change** - The change table represents an instance of a code change that is in the review system. The table also contains relevant information such as the author of the code change (`ch_authorId`).
- **Revision** - As a change gets reviewed, it may undergo several revisions of the source code before it is committed. The revision holds information, such as the final commit date of the code change (`rev_committedTime`).
- **People** - The people table was created to store all details of the review members. Each member has a unique id (`p_author Id`).
- **History** - The history table contains all messages or comments related to a review. The history table contains the messages attribute (`hist_message`) that can be used to identify all comments and activities related to the review process.
- **File** - The file table contains the details of the code changes. This table contains information such as the pathname (`f_filename`), and size (`f_linesInserted`, `f_linesDeleted`) of the code change.

Queries example. To utilize the data tables, we need to map the tables to the different aspects of the peer review process. Table 3 shows the rationale and hints to which attributes to use when making a query. A sample of other useful queries is available on our website. For example, to get all the core reviewers for a project we need to query the history table (`t_history`) for all people that have 1.) approved or 2.) reject a review or able to provide a score of either 3.) +2 or 4.) -2. This would be in the SQL query:

```
SELECT distinct hist_authorId FROM t_history
WHERE hist_message LIKE '%Looks good to me, approved%'
OR hist_message LIKE '%Code-Review+2%'
OR hist_message LIKE '%Do not submit%'
OR hist_message LIKE '%Code-Review-2%'
ORDER BY hist_createdTime ASC;
```

[5]https://code.google.com/p/gerrit (Feb 18, 2016)
[6]https://github.com/kin-y/miningReviewRepo/wiki/Database-Schema (Feb 18, 2016)

Table 3: Hints linking review concepts to our database schema

		Linked Tables	Rationale
People	id, name, email	People	The people table links to a unique member of the review community
	people roles	History	`History.hist_message` attribute is used to distinguish people types
	commit experience	Change, Revision	The # of `ch_Id` and `rev_Id` linked to a reviewer shows commit experience
	review experience	History, People	Count of `History.hist_changeId` shows review experience
Process	review states	Change	The `Change.ch_status` attribute shows the review states
	review voting	History	Review comments in `History.hist_message` attribute shows the voting results
	review period	Change, History	The difference of commit time (`rev_commit`) and review time (`ch_createdTime`) shows the review period
Product	code changes	Change, Revision, File	How many lines of code has changed in a file, a revision or a change
	revision info	Change, Revision	The revision table links to git commits in code review

5. RELATED PUBLICATIONS

There are several outputs generated from our research team, using both the old schema [8] and the updated one. The full listing of our publications can be found online at our website wiki[7]. One key contribution of our research team has been using the file patch of the patches to assist with the assignment of the most appropriate reviewer in the review process [18]. Thongtanunam et al. used the data to evaluate a reviewer recommendation approach called RevFinder. Another set of research has been into the study of the human factor with review teams with a creation of a social network (PeRSoN) [21, 22]. Related, we defined profiling metrics to categorize the review team member by their expertise types [10]. Another work involves a visualization of the core review dataset called the Review Data Analyzer (ReDA) [19]. Also, other researchers have used or expanded our core dataset [11, 16, 17].

6. REFERENCES

[1] https://android-review.googlesource.com (Feb 18, 2016).

[2] https://git.eclipse.org/r/ (Feb 18, 2016).

[3] https://gerrit.libreoffice.org (Feb 18, 2016).

[4] https://review.openstack.org (Feb 18, 2016).

[5] https://codereview.qt-project.org (Feb 18, 2016).

[6] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proc. ICSE '13*, pages 712–721, 2013.

[7] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[8] K. Hamasaki, R. G. Kula, N. Yoshida, A. E. C. Cruz, K. Fujiwara, and H. Iida. Who does what during a code review? datasets of oss peer review repositories. In *Proc. of MSR '13*, pages 49–52, 2013.

[9] O. Kononenko, O. Baysal, and M. W. Godfrey. Code review quality: How developers see it. In *Proc. of ICSE '16*, page to appear, 2016.

[10] R. G. Kula, A. E. C. Cruz, N. Yoshida, K. Hamasaki, K. Fujiwara, X. Yang, and H. Iida. Using profiling metrics to categorise peer review types in the android project. In *Proc. of ISSRE '12*, pages 146–151, 2012.

[11] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proc. of MSR '14*, pages 192–201, 2014.

[7]https://github.com/kin-y/miningReviewRepo/wiki/Publications (Feb 18, 2016)

[12] M. Mukadam, C. Bird, and P. C. Rigby. Gerrit software code review data from android. In *Proc. of MSR '13*, pages 45–48, 2013.

[13] D. L. Parnas and M. Lawford. The role of inspection in software quality assurance. *IEEE Trans. Softw. Eng.*, 29(8):674–676, 2003.

[14] P. C. Rigby, D. M. German, L. Cowen, and M.-A. Storey. Peer Review on Open-Source Software Projects. *ACM Transactions on Software Engineering and Methodology*, 23(4):1–33, 2014.

[15] G. Sabaliauskaite, F. Matsukawa, S. Kusumoto, and K. Inoue. Further investigations of reading techniques for object-oriented design inspection. *Information and Software Technology*, 45(9):571–585, 2003.

[16] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. Investigating code review practices in defective files: An empirical study of the qt system. In *Proc of MSR '15*, pages 168–179, 2015.

[17] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida. Revisiting Code Ownership and its Relationship with Software Quality in the Scope of Modern Code Review. In *Proc. of ICSE '16*, page to appear, 2016.

[18] P. Thongtanunam, C. Tantithamthavorn, R. Kula, N. Yoshida, H. Iida, and K. Matsumoto. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *Proc. of SANER 2015*, 2015.

[19] P. Thongtanunam, X. Yang, N. Yoshida, R. G. Kula, A. E. Camargo Cruz, K. Fujiwara, and H. Iida. Reda: A web-based visualization tool for analyzing modern code review dataset. In *Proc. of ICSME '14*, pages 605–608, 2014.

[20] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili. Detecting defects in object-oriented designs: Using reading techniques to increase software quality. In *Proc. of OOPSLA '99*, pages 47–56, 1999.

[21] X. Yang, R. G. Kula, C. C. A. Erika, N. Yoshida, K. Hamasaki, K. Fujiwara, and H. Iida. Understanding oss peer review roles in peer review social network (person). In *Proc. of APSEC '12*, pages 709–712, 2012.

[22] X. Yang, N. Yoshida, R. G. Kula, and H. Iida. Peer review social network (PeRSoN) in open source projects. *IEICE Transactions on Information and Systems*, E99-D(3):661–670, 2016.