Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2018) 26: 1343 – 1353

doi:10.3906/elk-1707-173

Will it pass? Predicting the outcome of a source code review

Çağdaş Evren GEREDE*, Zeki MAZAN
Department of Computer Engineering, Faculty of Engineering, TOBB University of Economics and Technology, Ankara, Turkey

Received: 14.07.2017 • **Accepted/Published Online:** 09.01.2018 • **Final Version:** 30.05.2018

Abstract: It has been observed that allowing source code changes to be made only after source code reviews has a positive impact on the quality and lifetime of the resulting software. In some cases, code review processes take quite a long time and this negatively affects software development costs and employee job satisfaction. Establishing mechanisms that predict what kind of feedback reviewers will provide and what revisions they will ask for can reduce the number of times this problem occurs. Thanks to such mechanisms, developers can improve the maturity of their code change requests before the review process starts. In this way, when they start the review, the process may advance quickly and more smoothly. In this study, as a first step towards this goal, we developed a mechanism to identify whether a change proposal would require any revisions or not for approval.

Key words: Source code review, revision prediction, mining software repositories

1. Introduction

Source code reviews are one of the important software development activities. They help with the early discovery of bugs, improve the readability and maintenance of source code, and have a positive impact on the quality and lifetime of software systems [1–4]. Today, it is widely common that in an open or closed source software project, any software change request (CR) is required to go through a code review process before being merged into the source code repository. In terms of how the team members collaborate, there are two ways of conducting code reviews [4]. The first one is called a synchronous review, which requires the author and inspectors to gather in a room and go through a more formal review process. The second one is called an asynchronous or "meetingless" inspection, and it does not require reviewers and inspectors to participate in the review process at the same time. As shown by Perpich et al. [5], in the past, asynchronous reviews were more cost-effective and at least as quality-effective as synchronous reviews. Today, when we examine popular open source software (OSS) code repository hosting sites such as github.com, bitbucket.com, and gitlab.com, we see that this is the only review model supported. This is not surprising, because OSS projects consist of temporally and geographically distributed contributors, and the asynchronous model allows them to collaborate more effectively. The OSS community has incorporated peer code review for quality assurance, and developers utilize dedicated code review tools such as Gerrit (https://www.gerritcodereview.com/), Codestriker (http://codestriker.sourceforge.net), and ReviewBoard (http://www.reviewboard.org) [6]. Similar observations have been reported in the literature for not only OSS but also closed source commercial software [7].

Despite the benefits of peer code reviews, when asynchronous reviews take a long time, they increase

*Correspondence: cegerede@etu.edu.tr

the software development time and can lead developers to become disgruntled. According to a study on the Linux kernel, which is an OSS project with more than 1000 contributors, only 30% of submitted patches (i.e. CRs) were accepted and a typical patch review can take weeks [8]. Especially for OSS projects, developers are physically far away from each other and often live in different time zones. In such situations, receiving even the first feedback can take several days. Anticipating how reviewers react to a CR, improving the CR based on this anticipation, and starting the official review process afterwards can shorten the duration of the review process. One simple strategy would be breaking up a big change into smaller consecutive proposals so that reviewers have less information to process. However, developers can only effectively apply such strategies after gaining considerable job experience and working on a project for a reasonable amount of time because a good knowledge of the team and project culture is important beforehand. As a result, having a tool that predicts how reviewers will react to a CR can help developers to improve their proposals so that they have a higher chance of going through a successful review process. The duration of the review processes can therefore be shortened, and the costs they add to the software development process can be reduced. In addition, being able to receive approvals with a lower number of review iterations can help developers to accomplish the same amount of work in less time, which can have a positive effect on developers' job satisfaction.

In this study, as a first step towards building a tool to help developers pass a code review, we seek answers to the following research question: can we identify whether a CR requires revisions or not to be approved?

If we can answer this question, we can stop a premature CR and allow the owner to revise the change. In order to be practically useful, in addition to providing the author with a simple yes/no answer, it is necessary to provide an explanation of why a CR would need revisions. This study does not yet reach the end goal of providing actionable feedback, but it does contribute to the state of the art in this direction.

2. Background and related work

2.1. Modern code review

In recent years, a more lightweight and less formal code review process has been adopted by many open source (e.g., Android, LibreOffice, and Eclipse) and proprietary (Google, Cisco, and Microsoft) projects [7]. Next we describe the Gerrit tool, a web-based standalone code review tool, and the typical review process driven by it. Figure 1 shows the Gerrit interface of a CR. The interface shows many properties of the CR such as the developer proposing the change (the owner), the developers reviewing the proposal (the reviewers), the name of the modified files, the bug that caused the change, the project and the branch of the repository the modified files belong to, the date, and a short summary of the modifications.

Figure 2 shows how Gerrit presents the modifications in each file in the CR as edit operations. To improve the comprehension of the changes proposed, the original and the updated text are displayed side by side using the result of a text edit distance algorithm [9,10]. Some visual effects are also applied to the side-by-side view to facilitate comprehension. For example, newly added lines are displayed with a green background. The message thread between the owner and the reviewers regarding a code block is shown on top of the related code block with a yellow background. The use of a diff-marked code listing, the use of visual effects, and the annotated source code blocks were first proposed in 1990s [5], and they are common among the available code review tools used today. A CR goes through three main states.

1) Open state: the owner creates a change, adds some reviewers to it, and sends it for review. In this state, the CR is accessible to any reviewer.

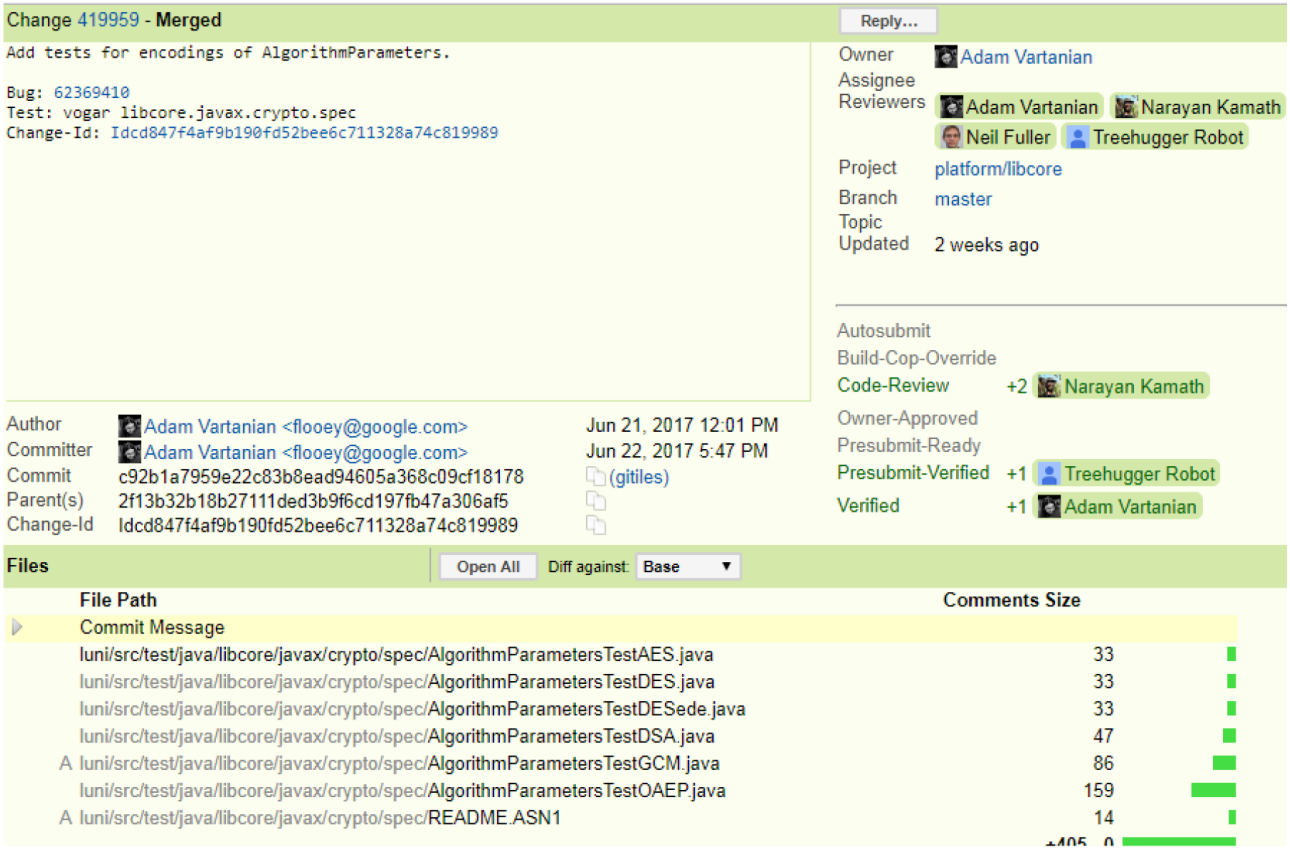

Figure 1. Screenshot from Gerrit (source: https://goo.gl/Qh7wuc).

2) Review state: reviewers begin to annotate proposed modifications with revision requests (see Figure 2). The owner then responds to each request with objections or with an updated CR with appropriate new code modifications. The CR stays in this state, while a back-and-forth communication continues between the owner and the reviewers until the reviewers give approval.

3) Decision state: once all requests from the reviewers are incorporated, the reviewers approve the CR. The CR is then merged into the code base. If a reviewer rejects a CR, then the CR is labeled as "abandoned". The verification and testing of the CR is also performed in this state and the positive outcome is a precondition to the approval.

2.2. Related work

Compared to all the related studies we describe in this section, we focus on a new problem, which is predicting whether or not a CR would be subjected to a revision request by any of its reviewers. The closest related body of work to this problem is predicting patch acceptance (see below). However, there is an important difference. For our problem, we need to differentiate between a patch that is accepted after one or more revisions and a patch that is accepted without any revisions.

Figure 2. Screenshot from Gerrit showing how original and proposed versions of a single file are displayed side by side (source: https://goo.gl/3MGPKs).

2.2.1. Mining review histories

As the code review datasets became publicly available via OSS projects, several studies were conducted to extract and analyze these datasets. Mukadam et al. [11] examined the general structure of the Android project's code review data hosted on Gerrit. Yang et al. [12] extended this work and brought together all of the data from five open source projects under the same representation, which allowed the analysis of data from three different aspects: people-related, process-related, and product-related. Hamasaki et al. [2] proposed an approach for downloading and processing such datasets in addition to publishing the data they collected from three projects (see http://sdlab.naist.jp/reviewmining/).

2.2.2. Exploratory studies

Several qualitative and quantitative exploratory studies were carried out to analyze various metrics to achieve more insights into code review processes. Rigby et al. [3,13] explored more than 10 projects' code review data and looked for answers to process related questions such as how long reviews take, how many people are involved in reviews, and how effective reviews are. In a follow-up study, Rigby et al. examined review policies of 25 OSS projects to understand how OSS peer review is different than traditional inspection. In order to better understand reviewers' behaviors during code reviews, Kitagawa et al. [14] attempted to define the code review process using a game theoretical model. Bird et al. [1] developed a system that computed key metrics about review processes so that development teams could improve their code review practices. Land et al. [15] studied the effects of inspector training on inspection performance and showed that when developers were trained with practice and working examples, their review performance improved the most. Gousios et al. [16] investigated the effects of the acceptance of pull requests in the pull-based development model and found developers' track record and test coverage as the main influencers.

2.2.3. Reviewer recommendations

In order to guide the reviewer selection process, Cunha et al. [17] investigated if there was a specific personality type that correlated with someone's ability to perform better code reviews. Ouni et al. [6] proposed a genetic algorithm-driven approach to identify the most appropriate peer reviewers, and they achieved 59% precision and 74% recall for accurately recommending reviewers. Kerzazi et al. [18] used social network analysis to map developers' editing, commenting, and reviewing activities into social networks by investigating if a sociotechnical analysis could help identify reviewers. Thongtanunam et al. [19] proposed a file location-based code-reviewer recommendation approach, leveraging the file path similarity of a previously reviewed file path to recommend an appropriate code reviewer.

2.2.4. Helping reviewers

There are also related studies focusing on improving reviewers' experience during code reviews. Zhang et al. [20] developed a patch inspection tool to help reviewers more easily understand the impact of a code change. Bosu et al. [21] studied the characteristics of useful reviews given by reviewers and developed a classification model that can distinguish whether certain reviewer feedback would be helpful to the review process.

2.2.5. Predicting patch acceptance

Helping developers with the code review process by providing feedback about the review outcome has been examined in a number of other studies. Jeong et al. [22] studied code review data from the Bugzilla system (https://www.bugzilla.org/) and attempted to predict the review outcome. Hellendoorn et al. [23] used language models to compute how similar a CR is to previous CRs. They then predicted whether the CR would be approved based on the review outcomes of similar CRs. Weißgerber et al. [24] analyzed how frequently patches sent via emails were accepted. A patch corresponds to a single revision in our study. They observed that only 40% of the patches were accepted and that a smaller patch had a higher chance of being accepted. Jiang et al. [8] conducted a study on the Linux kernel and examined the relationship between patch characteristics and patch reviewing/integration time. They were able to build prediction models for patch acceptance with 73% precision and recall.

3. Predicting patch revision

3.1. Dataset

We used the code review data produced during the development of the open source Android operating system. The Android project uses the Gerrit tool for code reviews, and the review history is available online (https://android-review.googlesource.com). The same data are also accessible via a REST-based (Representational State Transfer) API (application programming interface). The documentation for Gerrit's REST API is available online (https://gerrit-review.googlesource.com/Documentation/rest-api.html). The server responses are in JavaScript Object Notation format.

Using this REST API, Hamasaki et al. [2] collected the Android code review data for 11,633 code CRs made between 21 October 2008 and 27 January 2012. After applying basic transformations (e.g., the names of the reviewers are removed from the data while a column showing the number of reviewers per review is added), they made this dataset public to enable further research studies (https://goo.gl/jWJ5CA). We benefited from

this dataset in our study. In the Hamasaki dataset, there are 35 features for each review instance. The relevant features for our study can be categorized as follows:

1. Owner's track record: the number of the owner's CRs in the open/abandoned/merged/approved state; the number of proposals for which this user acted as a reviewer; the total number of messages the user produced during reviews.
2. Reviewers: the number of reviewers assigned to the review; the number of people that need to verify and validate the review process.
3. Context: the project name the change belongs to; the repository branch the change is being made on.
4. Change summary: the number of added/removed/changed lines and the number of added/deleted/modified files in the CR.
5. Result: whether the CR was approved or abandoned (CRs that were still open after a year were assumed to be implicitly abandoned).
6. Revision count: the number of revisions that took place (note that we use the feature named "#PatchSets" from the dataset for this and by the revision count we actually mean #PatchSets minus 1).
7. **Methodology**

In light of the terminology introduced above, we can formulate the problem needing to be worked on as follows: when a CR is given, how successfully can we predict whether the revision count is zero or more?

For an abandoned CR, we treat its revision count as infinite. To help the discussion, we define two target classes: **RevNone** (no revisions) and **RevSome** (some revisions). As a solution to this problem, we employ three different strategies. Each approach uses a different set of features to predict whether a CR belongs to RevNone or RevSome.

- **STRAGETY_ALL.** The first strategy uses all of the features except *Result* and *Revision count* to build a prediction model.
- **STRATEGY_CHANGE_SUMMARY**. In OSS projects, it is very common to have new contributors who do not have much of a track record (category 1). Therefore, there are cases where it is necessary to use a model that requires the owner to have a long history in the project. In addition, the change complexity is perceived as the main influencer of how long a review takes [25]. Based on these two observations, we wanted to examine how accurately we could make predictions if we only used *Change summary* (it does not include any information other than the modifications to the code and it is likely to have some correlation with change complexity.)
- **STRATEGY_CHANGE_LABELS.** The previous strategy uses very coarse-granular data to evaluate the change complexity such as number of lines added and number of files modified. In this strategy, we wanted to use finer-granular data to characterize the change complexity. We can say that not all of the changes are equivalent in perceived complexity. For instance, a proposal adding a one-line comment is more likely to be approved than a proposal adding a new parameter to a function. This kind of finer-granular data is not available in the dataset we used. In order to compensate for this, we benefited from a tool called ChangeDistiller, which was developed by Fluri et al. [26]. This

tool is an OSS (https://bitbucket.org/sealuzh/tools-changedistiller/wiki/Home) and works with Java programming language source code files. It runs a differencing algorithm on 'before' and 'after' versions of a file and summarizes each differing code block with one of 48 edit operation classes such as COMMENT_INSERT and PARAMETER_INSERT. The complete list of all operation classes is available online at https://goo.gl/KeCxce. For each CR, we downloaded all of the original and modified versions of the files inside the CR and ran ChangeDistiller to produce operation labels to describe the change. We then only used these labels to train our prediction model.

In our investigation, we used Weka (http://www.cs.waikato.ac.nz/ml/weka/), data mining software that supports many machine-learning algorithms. For all strategies, we first ran an automatic feature selection algorithm and then employed several machine-learning algorithms on the selected features to train our prediction models. We measured the prediction success of various trained models with 10-fold cross-validation. We used the following metrics for reporting the outcome performance:

- Accuracy: the number of CRs correctly categorized as RevNone or RevSome, divided by the total number of CRs;
- Precision: the number of CRs correctly categorized as RevNone, divided by the number of all CRs categorized as RevNone;
- Recall: the number of CRs correctly categorized as RevNone, divided by the number of all RevNone CRs.

When reporting our results in the next section, we use a ZeroR classifier as a baseline, which maps every CR to the majority class. In our dataset, 60% of the CRs are in the RevSome class. Therefore, the baseline accuracy for the first two strategies is 60%. For the last strategy, since ChangeDistiller only works on Java files, we had to eliminate the CRs without any Java files. This eliminated more CRs that are in the RevSome class, and the RevNone became the majority class at 65%. Therefore, the baseline accuracy rate for the third strategy was around 65%.

4. Results and discussion

4.1. STRATEGY_ALL

For this approach, the feature selection algorithms picked the following features as the most influential features determining the target classes (we give the actual feature name in parentheses):

- The number of CRs submitted by the owner so far (*Dev_#submitted*)
- The number of CRs merged by the owner so far (*Dev_#merges*)
- The number of CRs verified by the owner so far (*Dev_#verified*)
- The branch name the CR developed on (*Branch*)
- The number of reviewers (*AssignedReviewers*)
- The number of reviewers who are approvers (*Approvers*)
- The number of reviewers who are verifiers (*Verifiers*)

We next experimented with several machine-learning algorithms to train prediction models including a Bayesian network, SVM, AdaBoost, Bagging, and Random Forest. Among them, Random Forest performed the best. Table 1 shows the prediction performance. We see that the prediction performance is much higher than the ZeroR performance in all metrics. When we examined the classifications in detail, we concluded that the trained model correlates with the target class, mainly based on two categories of features:

Table 1. STRATEGY_ALL vs. ZeroR prediction performance.

Method	Accuracy	Precision	Recall	Class
Random Forest	94.59%	0.94/0.94	0.92/0.96	RevSome/RevNone
ZeroR	60%	0.60/0	1/0	RevSome/RevNone

1. The past code review statistics of the owner (the number of changes she made, the number of reviews she conducted, etc.), i.e. owner experience: as the values of the experience-related features increase, a "no revision" review is predicted (i.e. when the developer is more experienced, it is more likely for the change to be approved without any feedback).

2. The number of people involved in the code review process (reviewers, verifiers, etc.): as the number of people involved decreases, a "no revision" review is predicted (i.e. when a lower number of people is involved, it is more likely for the change to be approved without any revisions).

According to the above, when an inexperienced developer proposes a CR that requires many reviewers' participation, the model is likely to predict that the CR needs to be subjected to a revision.

4.2. STRATEGY_CHANGE_SUMMARY

For this strategy, there are six features available in the dataset we can leverage, which are the number of added/deleted/renamed files and the number of inserted/removed/modified lines. Similar to the previous approach, we achieved the best performance with the Random Forest algorithm. Table 2 shows the prediction success rate. The success rate can still be considered high, but it is nevertheless lower than what we achieved in the previous section. This is expected since we use a lower number of features for prediction. When we examine the results in detail, we observe that the prediction this time is correlated by the amount of the updated content. In other words, when there are many updated files and lines in a CR, the CR is likely to be categorized as RevSome.

Table 2. STRATEGY_CHANGE_SUMMARY vs. ZeroR prediction performance.

Method	Accuracy	Precision	Recall	Class
Random Forest	83%	0.78/0.86	0.79/0.85	RevSome/RevNone
ZeroR	60%	0.60/0	1/0	RevSome/RevNone

4.3. STRATEGY_CHANGE_LABELS

For this strategy, the following features were selected by the feature selection algorithm (in parentheses, we provide the actual change-type name as defined by ChangeDistiller):

- A statement is deleted (*STATEMENT_DELETE*)
- A statement is inserted (*STATEMENT_INSERT*)
- A statement is updated (*STATEMENT_UPDATE*)
- A statement's order is updated (*STATEMENT_ORDERING_CHANGE*)
- A statement's parent changed (*STATEMENT_PARENT_CHANGE*)
- A condition inside an expression, such as an if-statement, for-statement, while-statement, do-statement, changed (*CONDITION_EXPRESSION_CHANGE*)
- A comment is deleted (*COMMENT_DELETE*)
- A comment is inserted (*COMMENT_INSERT*)
- An else statement is added for an if statement (*ALTERNATIVE_PART_INSERT*)
- A field is added (*ADDITIONAL_OBJECT_STATE*)
- A method is added (*ADDITIONAL_FUNCTIONALITY*).

Table 3 shows the best performing prediction algorithm, AdaBoost, for this case. Unfortunately, it does not perform any better than the ZeroR model. Based on our analysis, we think that there are several reasons for this negative outcome. First, ChangeDistiller failed to assign any operation labels to many CRs. This could be because of bugs in the ChangeDistiller implementation or because of its categorization being limited. The ChangeDistiller source code is available as a GitHub repository, but we have not had the chance to extensively debug it yet. Second, we suspect that ChangeDistiller's edit operation labels do not have enough context to characterize and summarize the changes at an appropriate granularity. For example, the STATEMENT_INSERT edit operation class does not reflect where in the code the statement is added, how complex the statement itself is, what other code blocks depend on this statement, and what statements this statement affects. Even though our attempt to benefit from ChangeDistiller failed, we still believe that our intuition of utilizing the change complexity of a CR for revision prediction is promising. In the next section, we discuss potential avenues for further exploration.

Table 3. STRATEGY_CHANGE_LABEL vs. ZeroR prediction performance.

Method	Accuracy	Precision	Recall	Class
Ada Boost	64.84%	0.53/0.65	0.04/0.98	RevSome/RevNone
ZeroR	64.61%	0/0.64	0/1	RevSome/RevNone

5. Conclusion and future work

Enforcing code reviews has a positive impact on the quality and lifetime of software systems. In some cases, code reviews take a long amount of time to finish. This negatively affects developers' job satisfaction and increases software development costs. Predicting how reviewers react to a CR before the review process starts can help developers to further develop the CR and help them to go through review processes more smoothly. As a first

step towards this goal, we developed techniques to predict whether a CR would progress under revisions. The first prediction techniques we developed were guided by how experienced the author of the proposal was and the number of people involved in the review process. When we predicted using only the features related to the changed content, we observed that the prediction was correlated mostly by how big the changed content was. To make our prediction more sensitive to the changed content complexity, we labeled each CR with categories of edit operations identified by a tool called ChangeDistiller and used these labels for prediction. However, this approach did not perform as well as the first two. One potential avenue for future work is to improve ChangeDistiller implementation so that it is able to label all CRs with change types. Another avenue that can be explored is to incorporate ChangeDistiller's change-type significance categorizations into the prediction process. There are six "significance levels" as defined by ChangeDistiller: *None*, *Low*, *Medium*, *High*, and *Crucial*. Each change type is associated with a single significance level, and significance levels seem to be correlated with change complexity. For example, the deletion of a documentation code is considered to have no significance (*None*), while updating where a class inherits from is considered to have *Crucial* significance. For another direction, in order to represent change complexity, new features can be extracted from the proposed code modifications using the results from the literature on software complexity metrics. For instance, if a CR contains a new code block that has a high cyclomatic complexity, it may have a higher chance of being subjected to a revision. Finally, accompanying revision predictions with mechanisms that can provide actionable feedback to the proposal owner on the problematic code blocks that have a high probability of causing revisions is important to improve the code review process from a contributor's perspective.

References

[1] Bird C, Carnahan T, Greiler M. Lessons learned from building and deploying a code review analytics platform. In: Working Conference on Mining Software Repositories; 16–24 May 2015; Florence, Italy. New York, NY, USA: IEEE. pp. 191-201.

[2] Hamasaki K, Kula RG, Yoshida N, Cruz AEC, Fujiwara K, Iida H. Who does what during a code review? Datasets of OSS peer review repositories. In: Working Conference on Mining Software Repositories; 18–19 May 2013; San Francisco, CA, USA. New York, NY, USA: IEEE. pp. 49-52.

[3] Rigby PC, Bird C. Convergent contemporary software peer review practices. In: Foundations of Software Engineering; 18–26 August 2013; St. Petersburg, Russia. New York, NY, USA: ACM. pp. 202-212.

[4] Fagan M. Design and code inspections to reduce errors in program development. IBM Syst J 1976; 15: 182-211.

[5] Perpich JM, Perry DE, Porter AA, Votta LG, Wade MW. Anywhere, anytime code inspections: using the web to remove inspection bottlenecks in large-scale software development. In: International Conference on Software Engineering; 17–23 May 1997; Boston, MA, USA. New York, NY, USA: ACM. pp. 14-21.

[6] Ouni A, Kula RG, Inoue K. Search-based peer reviewers recommendation in modern code review. In: International Conference on Software Maintenance and Evolution; 2–7 October 2016; Raleigh, NC, USA. New York, NY, USA: IEEE. pp. 367-377.

[7] Bacchelli A, Bird C. Expectations, outcomes, and challenges of modern code review. In: International Conference on Software Engineering; 18–26 May 2013; San Francisco, CA, USA. New York, NY, USA: IEEE. pp. 712-721.

[8] Jiang Y, Adams B, German DM. Will my patch make it? And how fast? Case study on the Linux kernel. In: Working Conference on Mining Software Repositories; 18–19 May 2013; San Francisco, CA, USA. New York, NY, USA: IEEE. pp. 101-110.

[9] Canfora G, Cerulo L, Penta MD. Ldiff: An enhanced line differencing tool. In: International Conference on Software Engineering; 16–24 May 2009; Vancouver, Canada. New York, NY, USA: IEEE. pp. 595–598.

[10] Miller W, Eugene Myers, W. A file comparison program. Software Pract Exper 1985; 15: 1024-1040.

[11] Mukadam M, Bird C, Rigby PC: Gerrit software code review data from android. In: Working Conference on Mining Software Repositories; 18–19 May 2013; San Francisco, CA, USA. New York, NY, USA: IEEE. pp. 45-48.

[12] Yang X, Kula RG, Yoshida N, Iida H. Mining modern code review repositories: a dataset of people, process and product. In: Working Conference on Mining Software Repositories; 14–22 May 2016; Austin, TX, USA. New York, NY, USA: ACM. pp. 460-463.

[13] Rigby PC, German DM, Cowen L, Storey MA. Peer review on open-source software projects. ACM T Softw Eng Meth 2014; 23; 1-33.

[14] Kitagawa N, Hata H, Ihara A, Kogiso K, Matsumoto K. Code review participation: game theoretical modeling of reviewers in Gerrit datasets. In: Cooperative and Human Aspects of Software Engineering; 16 May 2016; Austin, TX, USA. New York, NY, USA: IEEE. pp. 64-67.

[15] Land LPW, Tan BCT, Bin L. Investigating training effects on software reviews: a controlled experiment. In: International Symposium on Empirical Software Engineering; 17–18 November 2005; Noosa Heads, Australia. New York, NY, USA: IEEE. pp. 356-366

[16] Gousios G, Pinzger M, Deursen AV. An exploratory study of the pull-based software development model. In: International Conference on Software Engineering; 31 May–7 June 2014; Hyderabad, India. New York, NY, USA: ACM. pp. 345-355.

[17] Cunha AD, Greathead D. Does personality matter? An analysis of code-review ability. Commun ACM 2007; 50; 109-112.

[18] Kerzazi N, Asri IE. Who can help to review this piece of code? In: Working Conference on Virtual Enterprises; 3–5 October 2016; Porto, Portugal. Berlin, Germany: Springer. pp. 289-301.

[19] Thongtanunam P, Tantithamthavorn C, Kula RG, Yoshida N, Iida H, Matsumoto K. Who should review my code? A file location-based code-reviewer recommendation approach for modern code review. In: International Conference on Software Analysis, Evolution, and Reengineering; 2–6 March 2017; Montreal, Canada. New York, NY, USA: IEEE. pp. 141-150.

[20] Zhang T, Song M, Pinedo J, Kim M. Interactive code review for systematic changes. In: International Conference on Software Engineering; 16–24 May 2015; Florence, Italy. New York, NY, USA: IEEE. pp. 111-122.

[21] Bosu A, Greiler M, Bird C. Characteristics of useful code reviews: an empirical study at Microsoft. In: Working Conference on Mining Software Repositories; 16–24 May 2015; Florence, Italy. New York, NY, USA: IEEE. pp. 146-156.

[22] Jeong G, Kim S, Zimmermann T, Yi K. Improving Code Review by Predicting Reviewers and Acceptance of Patches. Seoul, Korea: Research on Software Analysis for Error-free Computing Center, 2009.

[23] Hellendoorn V, Devanbu PT, Bacchelli A. Will they like this? Evaluating code contributions with language models. In: Working Conference on Mining Software Repositories; 16–24 May 2015; Florence, Italy. New York, NY, USA: IEEE. pp. 157-167.

[24] Weißgerber P, Neu D, Diehl S. Small patches get in! In: Working Conference on Mining Software Repositories; 10–11 May 2008; Leipzig, Germany. New York, NY, USA: ACM. pp. 67-76.

[25] Kononenko O, Baysal O, Godfrey MW. Code review quality: How developers see it. In: International Conference on Software Engineering; 14–22 May 2016; Austin, TX, USA. New York, NY, USA: ACM. pp. 1028-1038.

[26] Fluri B, Wuersch M, Pinzger M, Gall H. Change distilling: Tree differencing for fine-grained source code change extraction. IEEE T Software Eng 2007; 33: 725-743.