Automatic Classification of Review Comments in Pull-based Development Model

Zhixing Li, Yue Yu* , Gang Yin, Tao Wang, Qiang Fan, Huaimin Wang
College of Computer, National University of Defense Technology, Changsha, 410073, China
{lizhixing15, yuyue, yingang, taowang2005, fanqiang09, hmwang}@nudt.edu.cn

***Abstract*—The pull-based model, widely used in distributed software development, allows any contributor to fork a public repository, package contributions as a pull-request, and then merge back to the original repository. Code review is one of the most significant stages in pull-based development. It ensures that only high-quality pull-requests are accepted, based on the in-depth discussion among reviewers. Thus, automatically identifying what reviewers are talking about in the discussions is benificial to better understand the code review process.**

In this paper, we conduct a case study on three popular open-source software projects hosted on GitHub and construct a fine-grained taxonomy including 11 sub-categories for review comments. We then manually label over 5,600 review comments, and propose a Two-Stage Hybrid Classification (TSHC) algorithm to classify review comments automatically by combining rule-based and machine-learning techniques. Comparative experiments with a text-based method achieve a reasonable improvement on each project (9.2% in Rails, 5.3% in Elasticsearch, and 7.2% in Angular.js respectively) in terms of the weighted average F-measure.

***Index Terms*—Pull-request; code review; review comment;**

I. Introduction

The pull-based development model is becoming increasingly popular in distributed collaboration for open-source software (OSS) development [4], [8], [10]. On GitHub [1] alone, the largest social-coding community, nearly half of collaborative projects (already over 1 million [9] in January 2016) have adopted this model. In pull-based development, any contributor can freely *fork* (*i.e.*, clone) an interesting public project and modify the forked repository locally (*e.g.*, fixing bugs and adding new features) without asking for the access to the central repository. When the changes are ready to merge back to the master branch, the contributors submit a *pull-request*, and then a rigorous code review process is performed before the pull-request get accepted.

Code review is a communication channel where integrators, who are core members of a project, can express their concern for the contribution [10], [20], [22]. If they doubt the quality of a submitted pull-request, integrators make comments that ask the contributors to provide several use cases or improve the implementation. With pull-requests becoming increasingly popular, most large OSS projects allow for crowd sourcing of pull-request reviews to a large number of external developers [23], [24] concerned about the development of the corresponding project to reduce the workload of integrators. After receiving the review comments, the contributor usually responds positively and updates the pull-request for another round of review. Thereafter, the responsible integrator makes a decision to accept the pull-request or reject it by taking all judgments and changes into consideration.

Previous research has shown that code review, as a well-established software quality practice, is one of the most significant stages in pull-based development [17], [18], [23]. It ensures that only high-quality codes are accepted. Several approaches have been proposed to study on how the code review process influences pull-request acceptance [19], [25] and latency [22], and software release quality [17]. However, a few studies have systematically investigated the automatic identification of what reviewers are talking about in the review discussions which is benificial to better understand the code review process.

In this paper, we conduct a case study on three popular OSS projects hosted on GitHub to comprehensively understand the categories of review comments from various stakeholders. First, we construct a fine-grained taxonomy covering the typical motivations of reviewers in joining the review process. Second, we manually label a large set of review comments according to the defined categories. With this dataset, we propose TSHC, a two-stage hybrid classification algorithm that is able to automatically classify review comments by combining rule-based and machine-learning (ML) techniques. The key contributions of this study include the following:

- A fine-grained and multi-level taxonomy for review comments in the pull-based development model is provided in relation to technical, management, and social aspects.
- A high-quality manually labeled dataset of review comments, which contains more than 5,600 items, can be accessed via a web page [2] and used in further studies.
- A high-performance automatic classification approach for review comments is proposed. The approach leads to a significant improvement in terms of the weighted average F-measure, namely 9.2% in Rails, 5.3% in Elasticsearch, and 7.2% in Angular.js, compared with the text-based method.

The rest of this paper is organized as follows. Section II introduces the pull-based development model. Section III elucidates the approach of our study. Section IV elaborates

* Corresponding author.
DOI: 10.18293/SEKE2017-039
[1]https://github.com/

[2]https://www.trustie.net/projects/2455

the research result. Section V presents related work, and Section VI provides the conclusion and future work.

II. Preliminary

On GitHub, a growing number of developers contribute to the open source projects by using the pull-request mechanism [8]. As illustrated in Figure1, a typical contribution process based on pull-based development model on GitHub involves the following steps.

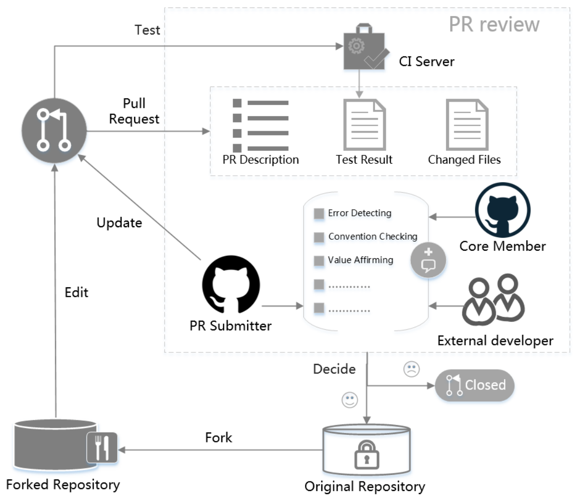

Fig. 1: Pull-based workflow on GitHub

Fork: Before contributing to an interesting project, the contributor has to fork the original project.

Edit: After forking, the contributor can edit locally without disturbing the main stream branch. He is free to do whatever he wants, such as implementing a new feature or fixing bugs according to the cloned repository.

Pull Request: When his work is finished, the contributor submits the changed codes from the forked repository to its source by a pull-request. Except for commits, the submitter needs to provide a title and description to explain the objective of his pull-request.

Test: Several reviewers play the role of testers to ensure that the pull-request does not break the current runnable state. They check the submitted changes by manually running the patches locally or through an automated manner with the help of continuous integration (CI) services.

Review: All developers in the community have the chance to review that pull-request in the issue tracker, with the existence of its description, changed files and codes, and test results. After receiving the feedback from reviewers, the contributor updates his pull-request by attaching new commits for another round review.

Decide: A responsible manager of the core team considers all the opinions of reviewers and merges or rejects the pull-request.

III. Approach

The goals of our work are to build a taxonomy for review comments in the pull-based development model and automate the comment classification according to the defined taxonomy.

A. Taxonomy Definition

Previous work has studied the challenges faced by pull-request reviewers and the issues introduced by pull-request submitters [10], [20]. Inspired by their work, we decide to comprehensively observe what reviewers are talking about in code reviews rather than merely focusing on technical and nontechnical perspectives.

We conducted a case study to determine the taxonomy scheme, which is developed manually through an iterative process of reading and analyzing review comments randomly collected from three projects hosted on GitHub. This process employs the following phases.

1) *Preparation phase*: The second author selects three projects (*i.e.*, Rails, Elasticsearch, and Angular.js) hosted on GitHub according to project popularity (*e.g.*, #star, #fork, and #contributor) and maturity (*e.g.*, project age, and pull-request usage). Table I shows the key statistics of the three projects in our dataset. Afterward, the first author randomly selects review comments from the three projects and labels each comment with a descriptive message.
2) *Execution phase*: The first author divides the previously labeled comments into different groups according to their descriptive messages. Unlike the aggregation process in card sorting [2], we initially set all comments to be in the same group and iteratively divide the group. Each comment is placed in the corresponding group(s). Through a rigorous analysis of existing literature and our own experience with working and analyzing the pull-based model in the last two years, we first identify the first level categories, which in turn are divided into other specific groups.
3) *Analysis phase*: By referring to the recognition result in the preceding phase, the second author abstracts taxonomy hierarchies and deduces general categories, which forms a two-level taxonomy scheme. The first and second authors, together with 10 other participants, verify the correctness and completeness of this scheme and make a final decision.

B. Two-Stage Hybrid Classification

To train the automatic classifier, we first manually labeled a set of review comments according to the defined taxonomy. For each project, we randomly sample 200 pull-requests (the comment count of which is greater than 0 and less than 30) per year from 2013 to 2015. Overall, 1,800 distinct pull-requests and 5,645 review comments are sampled. Based on this data set, we built TSHC which is illustrated by Figure 2. TSHC consists of two stages that utilize comments text and other information extracted from comments and pull-requests respectively.

TABLE I: Dataset of our experiments

Projects	Language	Application Area	Hosted_at	#Star	#Fork	#Contributor	#Pull-request	#Comment
Rails	Ruby	Web Framework	May. 20 2009	33906	13789	3194	14648	75102
Elasticsearch	Java	Search Server	Feb. 8 2010	20008	6871	753	6315	38930
Angular.js	JavaScript	Front-end Framework	Jan. 6 2010	54231	26930	1557	6376	33335

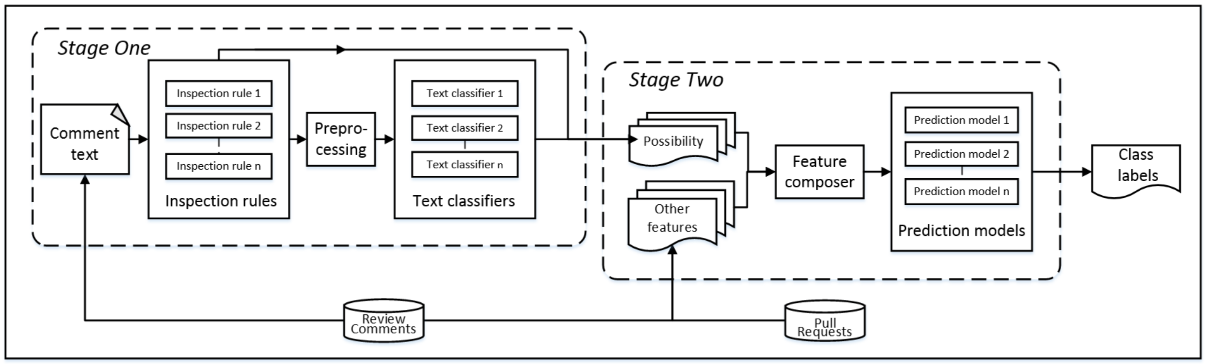

Fig. 2: Overview of TSHC

Stage One: The classification in this stage mainly utilizes the text part of each review comment and produces a vector of possibility (VP), in which each item is the possibility of whether a review comment will be labeled with the corresponding category.

Preprocessing is necessary before formal comment classification. Reviewers tend to reference the source code, hyperlink, or statement of others in a review comment to clearly express their opinion, prove their point, or reply to other people. These behaviors promote the review process but cause a great challenge to comment classification. Words in these reference texts contribute minimally to the classification and even introduce interference. Hence, we transform them into single-word indicators to reduce vocabulary interference and reserve reference information. Specifically, the source code, hyperlink, and statement of others are replaced with *'cmmcode'*, *'cmmlink'* and *'cmmtalk'* respectively.

After preprocessing, a review comment is classified by a rule-based technique, which uses inspection rules to match the comment text for each category. Several phrases often appear in the comments of a specific category. For instance, *"lgtm"* (abbreviation of *"looks good to me"*) is usually used by reviewers to express their satisfaction with a pull-request. This type of phrases are discriminating and helpful in recognizing the category of a review comment. Therefore, we establish an inspection rule for each category, which is a set of regular expressions abstracted from discriminating phrases. A category label is assigned to a review comment, and the corresponding item in VP is set to 1 if its inspection rule matches the comment text. The following shows examples of some regular expressions and the corresponding matched comments.

- Example 1
  - `(blank|extra) (line|space)s?`
  - *"Please add a new **blank line** after the include"*
- Example 2
  - `(looks|seem)s? (good|great|useful|awesome)`
  - *"**Looks good** to me, the grammar is definitely better."*
- Example 3
  - `(cc:?|wdyt|defer to|\br\?) (@\w+ *)+`
  - *"**/cc @fxn** can you take a look please?"*
- Example 4
  - `thanks?|thxs?|:(\w+)?heart:`
  - *"@xxx looks good. **Thank** you for your contribution **:yellow_heart:**"*

The review comment is then processed by the ML-based technique. ML-based classification is performed with scikit-learn[3], particularly the support vector machine (SVM) algorithm. The comment text is tokenized and stemmed to a root form [14]. We filter out punctuations from word tokens and reserve English stop words because we assume that common words play an important role in short texts, such as review comments. We adopt the TF-IDF (term frequencyinverse document frequency) model [3] to extract a set of features from the comment text and apply the ML algorithm to text classification. A single review comment often addresses multiple topics. Hence, one of the goals of TSHC is to perform multi-label classification. To this end, we construct text classifiers (TCs) for each category with a one-versus-all strategy. For a review comment which has been matched by inspection rule R_i (supposing that n categories of $C_1, C_2, \ldots, C_n$ exist), each TC ($TC_1, TC_2, \ldots, TC_n$), except for TC_i, is applied and predict the possibility of this review comment belonging to the corresponding category

Finally, the VP determined by inspection rules and text classifiers are passed on to *stage 2*.

Stage Two: The classification in this stage is performed on composed features. Review comments are usually short texts. Our statistics indicates that the minimum, maximum, and

[3]http://scikit-learn.org/

average numbers of words contained in a review comment are 1, 1527, and 32, respectively. The text information contained in a review comment is limited to be used to determine its category. Therefore, in addition to the VP generated in *stage one*, we also consider the following other features related to review comments.

`Comment_length`: This feature refers to the total number of characters contained in a comment text after preprocessing. Long comments are likely to argue about pull-request appropriateness and code correctness.

`Comment_type`: This binary feature indicates whether a review comment is inline comment or issue comment. An inline comment tends to talk about the solution detail, whereas an issue comment is likely to talk about other "high-level" issues, such as pull-request decision and project management.

`Core_team`: This binary feature refers to whether the comment author is a core member of a project or an external contributor. Core members are more likely to pay attention to pull-request appropriateness and project management

`Link_inclusion`: This binary feature identifies if a comment includes hyperlinks. Hyperlinks are usually used to provide evidence when someone insists on a point of view or to offer guidelines when someone wants to help other people.

`Ping_inclusion`: This binary feature refers to if a comment includes ping activity (occurring by the form of "@ username").

`Code_inclusion`: This binary feature denotes if a comment includes the source code. Comments related to the solution detail tend to contain source codes.

`Ref_inclusion`: This binary feature indicates if a comment includes a reference on the statement of others. Such a reference indicates a reply to someone, which probably reflects further suggestion to or disagreement with a person.

`Sim_pr_title`: This feature refers to the similarity between the text of the comment and the text of the pull-request title (measured by the number of common words divided by the number of union words).

`Sim_pr_desc`: This binary feature denotes the similarity between the text of the comment and the text of the pull request description (measured similarly as how `sim_pr_title` is computed). Comments with high similarity to the title or description of a pull-request are likely to discuss the solution detail or the value of the pull request.

Together with the VP passed from *stage 1*, these features are composed to form a new feature vector to be processed by prediction models. Similar to *stage 1*, *stage 2* provides binary prediction models for each category. In the prediction models, a new VP is generated to represent how likely a comment will fall into a specific category. After iterating the VP, a comment is labeled with class C_i if the i_{th} vector item is greater than 0.5. If all the items of the VP are less than 0.5, the class label corresponding to the largest possibility will be assigned to the comment. Finally, each comment processed by TSHC is marked with at least one class label.

IV. Result

A. Taxonomy

In the case study, we identified a fined-grained two-level taxonomy which is illustrated by Table II. There are four categories in the first level taxonomy, which are divided into more specific sub-categories respectively. The first category (*Correctness*) includes review comments which correct or imporve the quality of changed codes, while the second category (*Decision*) includes those that show reviewers' dicision of whether or not permitting the pull-request to merge back. In addition, reviewers are responsible to manage the review process (*Management*) and intertact with contributors (*Interaction*).

TABLE II: Complete taxonomy

Level-1	Level-2	Description
Correctness	Style	points out extra blank line, etc.
	Functionality	figures out functionality defect, etc.
	Test	demands submitter to provide test case, etc.
Decision	Approval	approves of the pull-request.
	Disagreeing	rejects to merge the pull-requestetc.
	Questioning	ask for more use cases .
Management	Roadmap	states the development roadmap , etc.
	Diversion	assigns other reviewers.
	Convention	ask for formulating commit messages, etc.
Interaction	Response	thanks for what other people do, etc.
	Encouragement	agrees with others' opinion, etc.

Although the first level taxonomy is more detailed than previous work, we refined it further in the second level taxonomy whose description can be seen in Table II. Moreover, Table III shows some of the example comments.

TABLE III: Example comments

Category	Example comments
Style	*"scissors: this blank line"*
Functionality	*"let's extract this into a constant. No need to initialize it on every call"*
Test	*"this PR will need a unit test, I'm afraid, before it can be merged"*
Approval	*"PR looks good to me. Can you ..."*
Disagreeing	*"I do not think this is a feature we'd like to accept."*
Questioning	*"Can you provide a use case for this change?"*
Roadmap	*"Closing as 3-2-stable is security fixes only now"*
Diversion	*"/cc @fxn can you take a look please?"*
Convention	*"Can you squash the two commits into one and also put [ci skip] in the commit message"*
Response	*"Thank you. This feature was already proposed and it was rejected."*
Encouragement	*":+1: nice one @cristianbica"*

B. Evaluation of TSHC

In the evaluation, we design a text-based classifier (TBC) as a comparison baseline. TBC uses the same preprocessing techniques and SVM models as used in TSHC. Classification performance is evaluated through a 10-fold cross validation, namely, splitting review comments into 10 sets, of which nine sets are used to train the classifiers and the remaining set is for the performance test. The process is repeated 10 times.

TABLE IV: Classification performance on Level-2 subcategories

Cat.	Rails						Elasticsearch						Angular.js					
	TBC			TSHC			TBC			TSHC			TBC			TSHC		
	Prec.	Rec.	F-M	Prec.	Rec.	F-M	Prec.	Rec.	F-M	Prec.	Rec.	F-M	Prec.	Rec.	F-M	Prec.	Rec.	F-M
Style	0.75	0.57	0.66	0.88	0.67	0.78	0.75	0.46	0.61	0.85	0.58	0.72	0.54	0.26	0.40	0.76	0.78	0.77
Functionality	0.63	0.84	0.74	0.71	0.86	0.79	0.67	0.92	0.80	0.77	0.82	0.80	0.65	0.71	0.68	0.69	0.71	0.70
Test	0.59	0.46	0.53	0.74	0.78	0.76	0.66	0.55	0.61	0.60	0.52	0.56	0.64	0.63	0.64	0.75	0.77	0.76
Approval	0.56	0.35	0.46	0.73	0.58	0.66	0.92	0.84	0.88	0.91	0.88	0.90	0.83	0.72	0.78	0.84	0.79	0.82
Disagreeing	0.50	0.26	0.38	0.63	0.43	0.53	0.65	0.36	0.51	0.80	0.67	0.74	0.63	0.48	0.56	0.61	0.51	0.56
Questioning	0.47	0.21	0.34	0.60	0.36	0.48	0.33	0.11	0.22	0.38	0.26	0.32	0.45	0.51	0.48	0.47	0.49	0.48
Roadmap	0.79	0.66	0.73	0.79	0.72	0.76	0.75	0.39	0.57	0.75	0.68	0.72	0.49	0.31	0.40	0.83	0.64	0.74
Diversion	0.83	0.77	0.80	0.98	0.78	0.88	0.57	0.35	0.46	0.88	0.90	0.89	0.35	0.16	0.26	0.96	0.67	0.82
Convention	0.83	0.76	0.80	0.90	0.81	0.86	0.94	0.57	0.76	0.96	0.76	0.86	0.85	0.76	0.81	0.83	0.82	0.83
Response	0.98	0.93	0.96	0.99	0.98	0.99	0.91	0.84	0.88	0.93	0.95	0.94	0.90	0.80	0.85	0.94	0.93	0.94
Encouragement	0.80	0.66	0.73	0.89	0.87	0.88	0.87	0.67	0.77	0.92	0.91	0.92	0.93	0.62	0.78	0.98	0.92	0.95
AVG	0.71	0.67	**0.69**	0.80	0.76	**0.78**	0.79	0.75	**0.77**	0.83	0.81	**0.82**	0.71	0.64	**0.67**	0.76	0.73	**0.75**

Table IV shows the precision, recall, and F-measure provided by different approaches for Level-2 categories. Our approach achieves the highest precision, recall, and F-measure scores in all categories with only a few exceptions.

To provide an overall performance evaluation, we use the weighted average value of F-measure [26] of all categories by the proportions of instances in that category. Equation 1 describes the formula to derive the average F-measure. In the equation, the average F-measure is denoted as F_{avg}, the F-measure of the i_{th} category as f_i, and the number of instances of the i_{th} category as n_i.

$$F_{avg} = \frac{\sum_{i=1}^{11} n_i * f_i}{\sum_{i=1}^{11} n_i} \tag{1}$$

The table indicates that our approach consistently outperforms the baseline across the three projects. Compared with that in the baseline, the improvement in TSHC running on each project in terms of the weighted average F-measure is 9.2% (from 0.688 to 0.780) in Rails, 5.3% (from 0.767 to 0.820) in Elasticsearch, and 7.2% (from 0.675 to 0.747) in Angular.js. These results indicate that our approach is highly applicable in practice.

We further study the review comments miscategorized by TSHC. An example is that *"While karma does globally install with a bunch of plugins, we do need the npm install because without that you dont get the karma-ng-scenario karma plugin."*. TSHC classifies it as belonging to *Functionality*, but it is actually a *Disagreeing* comment. The reason for this incorrect predication is twofold, namely, the lack of explicit discriminating terms and the too specific expression for rejection. Inspection rule of *Disagreeing* is unable to matched because of the lack of corresponding mathcing patterns. ML classifiers tend to categorize it into *Functionality* because the too specific expression of opinion makes it more like a low-level comment about code correctness instead of a high-level one about the pull-request decision.

We attempt to solve this problem by adding factors (*e.g.*, `Comment_type` and `Code_inclusion`) in *stage 2* of TSHC, which can help reveal whether a review comment is talking about the pull-request as a whole or the solution detail. Although the additional information improves the classification performance to some extent, it is not sufficient to differentiate the two types of comments. We plan to address the issue by extending the manually labeled data set and introducing a sentiment analysis.

V. Related Work

A. Code Review

Code review is employed by many software projects to examine the change made by others in source codes, find potential defects, and ensure software quality before they are merged [5], [12]. Traditional code review, which is also well known as the code inspection proposed by Fagan [7], has been performed since the 1970s. However, its cumbersome and synchronous characteristics have hampered its universal application in practice [2]. With the occurrence and development of VCS and collaboration tools, Modern Code Review (MCR) [16] is adopted by many software companies and teams. Different from formal code inspections, MCR is a lightweight mechanism that is less time consuming and supported by various tools. While the main motivation for code review was believed to be finding defects to control software quality, recent research has revealed that defect elimination is not the sole motivation. Bacchelli *et al.* [2] reported additional expectations, including knowledge transfer, increased team awareness, and creation of alternative solutions to problems.

B. Pull-request

Although research on pull-requests is in its early stages, several relevant studies have been conducted. Gousios *et al.* [8], [15] conducted a statistical analysis of millions of pull-requests from GitHub and analyzed the popularity of pull-requests, the factors affecting the decision to merge or reject a pull-request, and the time to merge a pull-request. Tsay *et al.* [19] examined how social and technical information are used to evaluate pull-requests. Yu *et al.* [25] conducted a quantitative study on pull-request evaluation in the context of CI. Moreover, Yue *et al.* [24] proposed an approach that combines information retrieval and social network analysis to recommend potential reviewers. Veen *et al.* [21] presented PRioritizer, a prototype pull-request prioritization tool, which works to recommend the top pull-requests the project owner should focus on.

C. Classification on Free Text

Several studies have been performed to analyze free text generated in the software development process. Antoniol *et al.* [1] conducted a survey on 1,800 issues from the BTS of three large open-source systems and concluded that the linguistic information contained in these issues is sufficient to distinguish "bug" issues from "non-bug" ones. Later on, Pingclasai *et al.* [13] used topic modeling to classify bug reports. Herzig *et al.* [11] conducted a fine-grained classification on 7,000 issue reports and analyzed the classification results of early research. Zhou *et al.* [26] proposed a hybrid approach by combining text-mining and data-mining techniques to automatically classify bug reports. Ciurumelea *et al.* [6] studied reviews on mobile apps, proposed an approach to automatically organize reviews according to predefined tasks (battery, performance, memory, privacy, etc.), and recommended the related source code that should be modified.

VI. Conclusion&Future Work

In this paper, we conduct a case study on three popular OSS projects hosted on GitHub, and construct a fine-grained taxonomy including 11 sub-categories for review comments. According to the defined taxonomy we manually label over 5,600 review comments and propose a two-stage hybrid classification algorithm to automatically classify review comments. The comparative experiment results show that our approach can return reasonably good results for most categories.

Nevertheless, TSHC performs poorly on a few Level-2 subcategories. More work could be done in the future to improve it. we plan to address the shortcomings of our approach by extending the manually labeled data set and introducing a sentiment analysis. Moreover, we will try to improve reviewer recommendation and pull-request prioritization based on the result in this paper.

Acknowledgment

The research is supported by the National Grand R&D Plan (Grant No. 2016-YFB1000805) and National Natural Science Foundation of China (Grant No.61502512, 61432020, 61472430, 61532004).

References

[1] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann Ga Neuc. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *Conference of the Centre for Advanced Studies on Collaborative Research, October 27-30, 2008, Richmond Hill, Ontario, Canada*, page 23, 2008.

[2] A Bacchelli and C Bird. Expectations, outcomes, and challenges of modern code review. In *International Conference on Software Engineering*, pages 712–721, 2013.

[3] Ricardo A Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. China Machine Press, 2004.

[4] Earl T Barr, Christian Bird, Peter C Rigby, Abram Hindle, Daniel M German, and Premkumar Devanbu. Cohesive and isolated development with branches. In *International Conference on Fundamental Approaches To Software Engineering*, pages 316–331, 2012.

[5] Olga Baysal, Oleksii Kononenko, and Reid Holmes. Investigating technical and non-technical factors influencing modern code review. *Empirical Software Engineering*, 21(3):1–28, 2016.

[6] Adelina Ciurumelea, Andreas Schaufelbhl, Sebastiano Panichella, and Harald Gall. Analyzing reviews and code of mobile apps for better release planning. In *IEEE International Conference on Software Engineering*, 2016.

[7] Michael E Fagan. Design and code inspections to reduce errors in program development. In *Pioneers and Their Contributions to Software Engineering*, pages 301–334. Springer, 2001.

[8] Georgios Gousios, Martin Pinzger, and Arie Van Deursen. An exploratory study of the pull-based software development model. In *International Conference Software Engineering*, pages 345–355, 2014.

[9] Georgios Gousios, Margaret Anne Storey, and Alberto Bacchelli. Work practices and challenges in pull-based development: the contributor's perspective. In *International Conference Software Engineering*, pages 285–296, 2016.

[10] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. Work practices and challenges in pull-based development: the integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering*, pages 358–368. IEEE, 2015.

[11] Kim Herzig, Sascha Just, and Andreas Zeller. *It's not a Bug, it's a Feature: How Misclassification Impacts Bug Prediction*. 2013.

[12] Shane Mcintosh, Yasutaka Kamei, and Bram Adams. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):1–44, 2016.

[13] N. Pingclasai, H. Hata, and K. I. Matsumoto. Classifying bug reports to bugs and other requests using topic modeling. In *Asia-Pacific Software Engineering Conference*, pages 13 – 18, 2013.

[14] M. F Porter. *An algorithm for suffix stripping*. Morgan Kaufmann Publishers Inc., 1997.

[15] Peter C Rigby, Alberto Bacchelli, Georgios Gousios, and Murtuza Mukadam. *A Mixed Methods Approach to Mining Code Review Data: Examples and a study of multi-commit reviews and pull requests*. 2014.

[16] Peter C. Rigby and Margaret Anne Storey. Understanding broadcast based peer review on open source software projects. In *International Conference on Software Engineering*, pages 541–550, 2011.

[17] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. Investigating code review practices in defective files: an empirical study of the qt system. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 168–179. IEEE Press, 2015.

[18] Patanamon Thongtanunam, Shane Mcintosh, Ahmed E. Hassan, and Hajimu Iida. Review participation in modern code review. *Empirical Software Engineering*, pages 1–50, 2016.

[19] Jason Tsay, Laura Dabbish, and James Herbsleb. Influence of social and technical factors for evaluating contribution in github. In *ICSE*, pages 356–366, 2014.

[20] Jason Tsay, Laura Dabbish, and James Herbsleb. Let's talk about it: evaluating contributions through discussion in github. In *The ACM Sigsoft International Symposium*, pages 144–154, 2014.

[21] Erik Van Der Veen, Georgios Gousios, and Andy Zaidman. Automatically prioritizing pull requests. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 357–361. IEEE Press, 2015.

[22] Yue Yu, Huaimin Wang, Vladimir Filkov, Premkumar Devanbu, and Bogdan Vasilescu. Wait for it: Determinants of pull request evaluation latency on github. In *MSR*, 2015.

[23] Yue Yu, Huaimin Wang, Gang Yin, and Charles X Ling. Reviewer recommender of pull-requests in github. In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 609–612. IEEE, 2014.

[24] Yue Yu, Huaimin Wang, Gang Yin, and Tao Wang. Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment? *Information and Software Technology*, 74:204–218, 2016.

[25] Yue Yu, Gang Yin, Tao Wang, Cheng Yang, and Huaimin Wang. Determinants of pull-based development in the context of continuous integration. *Science China Information Sciences*, 59(8):1–14, 2016.

[26] Yu Zhou, Yanxiang Tong, Ruihang Gu, and Harald Gall. Combining text mining and data mining for bug report classification. *Journal of Software: Evolution and Process*, 2016.