Towards Prioritizing GitHub Issues

Akash Balasaheb Dhasade, Akhila Sri Manasa Venigalla, Sridhar Chimalakonda
Research in Intelligent Software & Human Analytics (RISHA) Lab
Indian Institute Of Technology
Tirupati, India
akashdhasade@gmail.com,{cs18m017,ch}@iittp.ac.in

ABSTRACT

The vast growth in usage of GitHub by developers to host their projects has led to extensive forking and open source contributions. These contributions occur in the form of issues that report bugs or pull requests to either fix bugs or add new features to the project. On the other hand, massive increase in the number of issues reported by developers and users is a major challenge for integrators, as the number of concurrent issues to be handled is much higher than the number of core contributors. While there exists prior work on prioritizing pull requests, in this paper we make an attempt towards prioritizing issues using machine learning techniques. We present the *Issue Prioritizer*, a tool to prioritize issues based on three criteria: issue lifetime, issue hotness and category of the issue. We see this work as an initial step towards supporting developers to handle large volumes of issues in projects.

CCS CONCEPTS

• **Software and its engineering** → *Open source model*; • **Computing methodologies** → Machine learning; Natural language processing.

KEYWORDS

Automatic Issue Prioritisation, GitHub Issues, Multiple Concurrent Issues, Priority Ranking, Dynamic Tracking

ACM Reference Format:
Akash Balasaheb Dhasade, Akhila Sri Manasa Venigalla, Sridhar Chimalakonda. 2020. Towards Prioritizing GitHub Issues. In *13th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference) (ISEC 2020), February 27–29, 2020, Jabalpur, India.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3385032.3385052

1 INTRODUCTION

Social coding in the present day has become a common practice with the presence of code sharing platforms such as GitHub, BitBucket and SourceForge. GitHub is one of the most commonly used and the largest platform that hosts open source projects [4] with about 100 million repositories[1] and 37 million users[2]. It comprises of more than 33 million public repositories[3], thus supporting developers to share their code and to get insights and contributions from other developers or practitioners across the world, without limitations of location or affiliation [3]. It provides various functionalities such as hosting source code, issue reporting and tracking, suggesting enhancements and resolving issues, continuous integration workflows with version control and so on [10, 12, 16].

Issues are user reports that are reported when a problem or deviation from expected behaviour is observed [1]. Issue reports are used to track tasks, enhancements and bugs for software projects on GitHub. Issues on GitHub define some of the milestones of a project and are primarily characterized by their titles, description, labels, assignees and comments[4]. Several public repositories on GitHub have many open issues to be resolved by integrators. For example, tensorflow repository has more than 2500 open issues[5]. The current GitHub interface allows developers to sort open issues based on multiple criteria such as age of issue, updated status of issues and so on. It is observed that the integrators face challenges in prioritizing work, in view of selecting the issues to be addressed first during concurrent issues [3]. Researchers have proposed metrics based on user profile to prioritize issues [11]. However, a better prioritisation demands additional features. In this paper, we present the design and implementation of the *Issue Prioritizer*, a tool to prioritize issues on GitHub. The *Issue Prioritizer* examines all open issues and presents top issues that require immediate attention to the integrator. Dynamically reacting to real time changes in issue-state, it acts as a priority inbox for issues.

2 RELATED WORK

Large availability of data on GitHub, such as contributor information, popularity of a repository, count and information of issues, pull requests and so on, has facilitated researchers to perform several studies in analysing various trends in the domain of software engineering [8, 14, 15]. Guzman et al. have studied the relation of nature of commit messages with programming language used, popularity of repository and several other factors [6]. Pham et al. have observed that labelled issues were solved quicker by analysing about 3M repositories [2].

Researchers have also proposed various tools and approaches to help developers use GitHub [7, 9, 13, 17]. A *reviewer recommender system* to predict the most relevant reviewer to whom the pull request could be assigned for review when a pull request is lodged

ISEC 2020, February 27–29, 2020, Jabalpur, India

ACM ISBN 978-1-4503-7594-8/20/02…$15.00
https://doi.org/10.1145/3385032.3385052

[1] https://venturebeat.com/2018/11/08/github-passes-100-million-repositories/
[2] https://github.com/search?q=type:user&type=Users
[3] https://github.com/search?q=is:public
[4] https://guides.github.com/features/issues/
[5] https://github.com/tensorflow/tensorflow

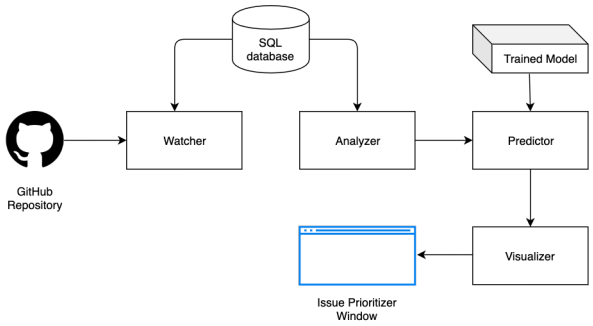

Figure 1: *Issue Prioritizer* Architecture

on GitHub has been proposed [17]. *PRioritizer* recommends pull requests to be focused on, to the project owner, by considering both static information and dynamic information such as prior actions on pull requests [13]. An approach to predict the life-time of an issue, in view of *whether an issue could be closed in a given calendric period*, based on dynamic and contextual features of the issue and repository has also been proposed [9]. *GiLA* supports users to analyse issues of a repository, through visualizations that are generated based on label based categorization of issues in the project [7]. Noei et al. have proposed an approach to identify the issues that need immediate attention by mapping them to related user reviews [11] . They have suggested that the issues being mapped to highest number of user reviews needed immediate attention as this mapping could have implied the most important feature to be fixed in a project.

Prioritizing issues and pull requests could be considered to be interlinked, as most of the pull requests are aimed to address open issues in a repository. Gousios et al. have analyzed work practices and challenges in GitHub based projects from an integrators' view and found out that integrators prioritize contributions by examining their criticality (in case of bug fixes), their urgency (in case of new features) and their size [5]. A pull request is crucial if it is attempting to fix a bug that is causing crashes, so it needs to be examined first. If the pull request is adding a new feature to the tool it is important but not as important as the former. This rationale emphasizes the vital role played by category of the pull request in deciding it's priority. *PRioritizer* [13] does not consider the liveness and asynchrony of pull requests. Essentially, high recent activity on the issue could be an indicative of trending issue. The approach proposed by Noei et al. in [11] deals only with user reviews, however, importance of an issue is observed to be determined by other factors such as issue lifetime and issue category. Hence, we propose the *Issue Prioritizer*, a tool that prioritizes issues based on three criteria that we identified – *issue lifetime*, *issue hotness* and *the category of issue.*

3 ARCHITECTURE OF ISSUE PRIORITIZER

The *Issue Prioritizer* follows a loosely coupled architecture similar to the *PRioritizer*[13] as shown in Figure 1, with the components:

Watcher- It is responsible for keeping track of new issues and state changes in the old issues. It maintains a time stamp of last updated time for every issue. When the *Issue Prioritizer* is setup for the first time, the watcher fetches all issues and stores them in a SQL database. This storage of data is crucial for quick retrieval of issues in later stages of priority calculation. After the setup, when watcher is called, it scrapes only those issues which have been modified/added/closed after the time stamp and updates the respective state in the SQL database. All data retrieval is done using GitHub API v3.

Analyzer- It is responsible for extracting the required data from SQL database of issues and evaluating metrics required for priority calculation.

Predictor- The metrics evaluated by the analyzer along with other required data are passed to the predictor which uses the trained ML model to predict the probable closing date of the issues. It then evaluates the final priorities of all open issues and passes the list of prioritized issues to the visualizer.

Visualizer- The visualizer presents the list of issues, enriched with information extracted from the analysis and prediction phases, as shown in Figure 2.

Table 1: Issue features and their nature

Feature	Description	Type
Author of issue	Is the author a project member?	Static
Comments	Number of discussion comments	Dynamic
Assignees	No of assignees to look over the issue	Static
Title and description	The text associated with issue title and description	Static
Labels	The type and number of labels assigned to the issue	Static
Age	Days between creation of issue and current time	Dynamic

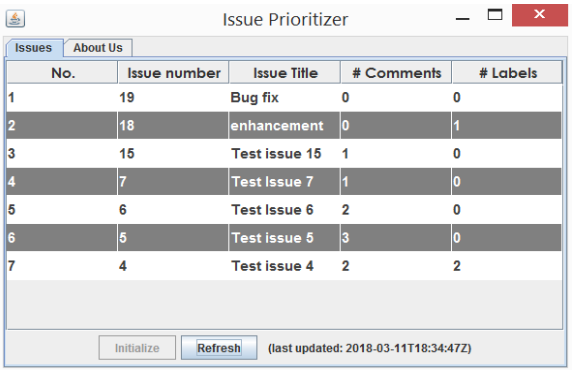

Figure 2: First view of *Issue Prioritizer*

4 ISSUE FEATURES & PRIORITY SCHEME

An issue is characterized by several features such as the *author of the issue*, *comments on the issue*, *assignees*, *title*, *body*, *labels* and the *age of the issue*. Though the list is not exclusive, it represents the most critical features for priority calculation. An issue raised by the owner of the repository is likely to receive immediate attention over another issue raised by a non-member. Similarly, a trending issue is likely to have received many comments in a shorter span. *Labels* on an issue help developers to group and address issues. The *title* and the *body of the issue* contain rich text information to help identify category of the issue when not labeled. Likewise,

the *number of assignees* for an issue is indicative of urgency of fix. Developers also deem the *age* of issue to be important, which is measured as time elapsed since the issue creation. Very old issues are likely to be addressed last.

To arrive at an appropriate model for issue prioritization, we further classify features as being static and dynamic. The features whose values once specified do not change (or hardly change) with time, such as *assignees*, are considered to be static features while those that keep on changing (or frequently change), such as the comments, are considered to be dynamic features. Assignees once assigned by the developer look over the issue throughout its lifetime but might change in rare cases. *Comments* are a good measure of recent activity on the issue and keep changing frequently with time. A summary of all features and their nature is presented in Table 1. The Issue Prioritizer identifies top priority issues based on 3 criteria: *issue hotness*, *predicted close date of issues* and *category of the issue*. Each of these criteria utilizes a subset of the features in Table 1.

4.1 Hotness of the Issue

An issue undergoes several events during its lifespan. The most common events are *commented, labeled, unlabeled, assigned* and *unassigned* amongst many others[6]. Such an event results in a change of state for the issue, for example, a change in *number of assignees* after being *assigned*. The new state then triggers the next event as this continues over the lifetime of issue. The hotness of an issue is a measure of the recent activity on the issue. We draw the parameters for modelling issue hotness from the model used by YOUTUBE[7] for identifying trending videos. Table 2 presents an analogy between both the models.

Table 2: Model similarity - Youtube trending videos & *Issue Prioritizer*

Youtube trending vidoes	*Issue Prioritizer*
Age of video	Age of issue
Growth rate of views	Growth rate of comments
Where are the views coming from?	Author of comment
View count	Comment and label count

Thus, the following four features are used for measuring issue hotness: *comments, labels, assignees* and *author association*. As discussed earlier, the issues raised by project owners are likely to draw more attention than non-members. While this is true in the initial stages of the issue, the effect of association of author in influencing issue hotness deteriorates with the life of the issue. This deterioration is also true of other features, an old comment for instance, is less likely be the cause of recent activity on the issue. To model this influence correctly, the formula for issue hotness entails a reduction in contribution of each instance of the feature governed by the elapsed time. The factor for reduction is determined by the nature of the issue. Thus, dynamic features contribute through a factor of $log_2(elapsed\_time)$ while the static features contribute through a factor of $log_{10}(elapsed\_time)$ indicating a slow deterioration with respect to elapsed time as compared to dynamic features. If t_0 denotes the time of creation of the respective feature and t denotes the current time, the formula for issue hotness is stated as,

$$\textbf{hotness} = \sum \frac{w_{comment}}{log_2(2+(t-t_0))} + \sum \frac{w_{assignees}}{log_{10}(2+(t-t_0))} + \frac{authAssoc}{log_{10}(2+(t-t_0))} + \sum \frac{w_{label}}{log_{10}(2+(t-t_0)))}$$

where the w_x's represent the weight of feature x. We consider $w_x = 1$ for all x in our evaluations but can be customized subject to requirements.

[6]https://developer.github.com/v3/issues/timeline/
[7]https://support.google.com/youtube/answer/7239739?hl=en

Table 3: Close time class

Days Class	Number
0 - 2	Class 0
3 - 7	Class 1
8 - 14	Class 2
15 - 28	Class 3
> 28	Class 4

Table 4: Category wise priority assignments

Category of the Issue	Priority
fix	3
error	2
update	1

4.2 Lifetime of the issue

We predict the lifetime of issue in days using three of the above features: *number of assignees, number of comments* and *author associated with the issue*. Choice of these features from all available set of features is intuitive from the fact that more the number of assignees, faster the issue being closed. Likewise, an issue with author as a project member is more likely to get closed than the issues with non-member authors. The choice of these features is not exclusive but is subject to trial. The closing time of the issue was categorized into five classes having analyzed the plot of number of issues vs closing time of issues for the tensorflow repository on GitHub[8]. The plot is presented in Figure 3. The choice of tensorflow repository for our evaluations is discussed in §5. We observe that a lot of issues are resolved within first 2 days of creation while the maximum that any issue was not resolved was around 80 days. Table 3 presents 5 different classes with associated closing times in terms of number of days.

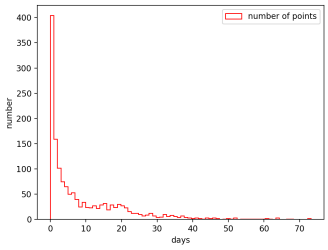

Figure 3: Lifetime vs Frequency of Issues

4.3 Category of the issue

An issue can be categorized into different categories such as bugfix, feature addition, error and other repository specific categories. This information is crucial since an issue labeled as a bug that causes

[8]https://github.com/tensorflow/tensorflow

the system to crash needs to be addressed earlier than the one that demands feature addition. Developers on GitHub can assign labels to the issues. Since this requires considerable time from the developer to read through the title and body of the issue, we find many unlabeled issues in repositories. Also, the labels are developer specific - *cla:yes*, *TF 2.0*, *comp:lite* are some examples of labels from the tensorflow repository. This makes it difficult to withdraw issue category information even from the labeled issues and demands other techniques. We use Latent Dirichlet Allocation (LDA) to categorize the issues. It builds a topic model per document (issue in our case) and a word model per topic. Thus, it can classify every issue into one of the categories based on the issue *title* and *body*. We create a corpus of text associated with every issue by appending the *title* and the *body*. On applying LDA on the text corpus, we get the following three topic models - (1),(2) and (3). We then associate a category to each topic along with a priority ordering as presented in Table 4. Some of the topics turned out to be tensorflow specific. This is because LDA was trained only on tensorflow issues, which could be fixed by considering a larger dataset of mixed repositories. We categorize every issue into one of these categories using the LDA model and utilize the category in final priority calculation.

$$(0,' 0.044 * "fix" + 0.008 * "mkl"') \tag{1}$$

$$(1,' 0.017 * "branch" + 0.014 * "update"') \tag{2}$$

$$(2,' 0.025 * "tensorflow" + 0.017 * "error"') \tag{3}$$

4.4 Final priority calculation

The final priority is the weighted sum of its components as shown,

$$\textbf{priority} = \frac{W_{closeTime}}{(closeTime + 1)} + W_{hotness} * hotness + W_{category} * CategoryPriority$$

Some teams may prefer using hotness of the issue as the most important criteria in deciding which issue to address first while other teams may deem category of the issue as the most important. While this is subjective, a customization through weights promotes flexibility and supports user interest. For ex. the first team mentioned above may use a combination of $\{W_{closeTime} = 0.2, W_{category} = 0.2, W_{hotness} = 0.6\}$. The tool can thus be personalized based on the requirement of the user. We evaluate two of three criteria in the following section and discuss several challenges.

5 EVALUATION

5.1 Dataset

We scraped 1500 closed issues and 1500 open issues from the tensorflow repository, following a qualitative case study approach to evaluate the proposed criteria with tensorflow as case study. The choice of tensorflow repository is due its to active development and wide usage by the ML community. It is representative of a repository with large number of issues (~ 3000 open issues, ~ 19000 closed issues) and few contributors (~ 2400) as compared to number of issues. The presented numbers were taken at the time of writing this work (Dec 2019). All the data retrieval is done using GitHub API v3 and the features mentioned in Table 1 were extracted for every issue.

5.2 Hotness of the issue

In order to evaluate the criterion for issue hotness, we create a series of issue states from the issue timeline[9]. An issue undergoes several events which result in a change of state during its lifespan. We consider five events, each of which alter one of features used for measuring hotness - *commented, labeled, unlabeled, assigned, unassigned*. A sample timeline for an issue in presented in Figure 4.

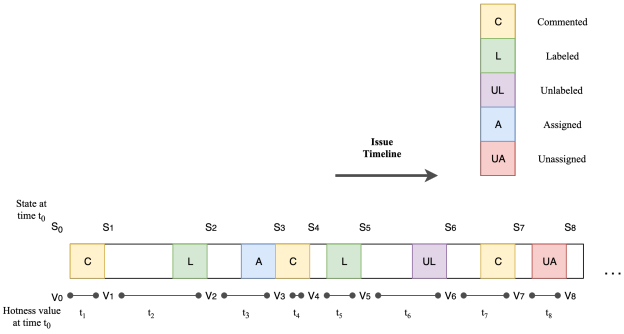

Figure 4: Example Issue Timeline

S_i's represent the states of the issues after every event. V_i's represents the issue hotness at the given point of time as measured by the proposed formula. The t_i's represent the time elapsed until the occurrence of next event. A trending issue is expected to have a higher value of hotness and a lower time elapsed until the occurrence of next event. We thus evaluate the correlation coefficient between V and t. The dataset of 3000 issues (1500 open and 1500 closed) results in 33062 states. Some issue states lead to $t_i = 0$. This is usually the case when two or more labels are added at same time or two or more assignees are assigned but are registered as different events by GitHub. We drop all issue states with $t_i = 0$ to measure unbiased coefficients. This results in 24633 issues states in total.

We obtain a Pearson correlation coefficient of -0.2328 and a Spearman correlation coefficient of -0.5535 between V and t for the above dataset. This suggests a good negative relationship between the measured hotness and the time for next event. Higher the issue hotness, lower should be the time until next event. We currently include five of the total events that occur on any issue. An inclusion of more events and extended feature set can be expected to improve the correlation coefficients.

5.3 Lifetime of the issue

We have trained 5 different ML models on the dataset of closed issues, and obtained accuracies as shown in the Table 5. We observed that decision tree classifier and random forest classifier performed better than other classifiers with accuracy of 48%. The results are biased towards the majority class which is class 0. While this is a preliminary evaluation on a single case study, an exploratory study of features affecting issue close times and training on larger dataset is required to improve our results.

[9]https://developer.github.com/v3/issues/timeline/

Table 5: ML models and their accuracies

Classifier	Accuracy (%)
K Nearest Neighbor (n = 29)	41.33
Logistic Regression	47.33
Decision tree (max depth 5)	48.00
Random Forest	48.00
SVM	46.66

5.4 Challenges in model training and evaluation

The priority of issue can be subjective in most cases and is heavily influenced by developer interest. It is difficult to assign absolute priority numbers on a fixed scale to every issue since different developers may assign different priority numbers to the same issue. This lack of absolute priority numbers prevent an end-to-end training of ML models using the three criteria. A generalized priority scheme accepted by all developers presents a good future research direction to create such datasets for training ML models. The challenge in evaluation of LDA model for identifying category of issues is again the lack of dataset of categorized issues. As discussed earlier, the assigned labels on issues can be developer or repository specific and do not necessarily suggest the category of the issue. A manual labeling of issues into a definite proposed set of categories would be required to evaluate category prediction models.

6 LIMITATIONS

This work is only a preliminary step towards the direction of prioritizing GitHub issues. As such, the first cut evaluation of the tool was done by the authors and few students as part of Software Engineering course. We asked students to evaluate the priority scheme on their projects and found that the ordering seemed relevant to most students. However, we plan to do a systematic evaluation of full scheme in the next revision. The current evaluations based only on tensorflow repository and the unevaluated category classification of issue present a threat to validity to our approach. A small and unevenly balanced dataset currently used for issue lifetime prediction is subject to further analysis for accurate results. The user interface of the tool is also preliminary and we plan to make this tool as a plugin to GitHub.

7 CONCLUSION & FUTURE WORK

In this paper, we proposed the idea of prioritizing issues in GitHub repositories and presented the *Issue Prioritizer* to demonstrate this idea. *Issue Prioritizer* considers various static and dynamic features to prioritise issues based on 3 criteria- *Hotness of an issue, Issue Lifetime* and *Category of the issue*. A dataset of 3000 issues was extracted for evaluation of the *Issue Prioritizer*. Each criteria uses a subset of the features to arrive at a value used in final calculation of priority. The hotness formula was evaluated to have a spearman correlation coefficient of -0.5535 and decision tree was found to be the best model for predicting issue close time with an accuracy of 48%. 3 categories of issues have been identified using the LDA model. The issues are then labelled based on the outcome of the model. For a given repository, *Issue Prioritizer* lists open issues in order of their priorities, based on customized weights assigned by user for each criteria. Thus, *Issue Prioritizer* offers personalization to the users. It is expected to reduce the burden of prioritizing issues to be addressed in the face of multiple concurrent issues.

As a future work, we plan to improvise *Issue Prioritizer* by improving ML model that incorporates study of features which affect the closing time of an issue. We also plan to introduce *Issue Prioritizer* as a Github plugin. Categories being classified by Issue Prioritizer could be systematically evaluated and the dataset could be extended to include more features and issues from other repositories for better results.

REFERENCES

[1] Tegawendé F Bissyandé, David Lo, Lingxiao Jiang, Laurent Réveillere, Jacques Klein, and Yves Le Traon. 2013. Got issues? who cares about it? a large scale investigation of issue trackers from github. In *2013 IEEE 24th international symposium on software reliability engineering (ISSRE)*. IEEE, 188–197.

[2] Jordi Cabot, Javier Luis Cánovas Izquierdo, Valerio Cosentino, and Belén Rolandi. 2015. Exploring the use of labels to categorize issues in open-source software projects. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 550–554.

[3] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social coding in GitHub: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on computer supported cooperative work*. ACM, 1277–1286.

[4] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. 2014. Lean GHTorrent: GitHub data on demand. In *Proceedings of the 11th working conference on mining software repositories*. ACM, 384–387.

[5] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. 2015. Work practices and challenges in pull-based development: the integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 358–368.

[6] Emitza Guzman, David Azócar, and Yang Li. 2014. Sentiment analysis of commit comments in GitHub: an empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 352–355.

[7] Javier Luis Cánovas Izquierdo, Valerio Cosentino, Belén Rolandi, Alexandre Bergel, and Jordi Cabot. 2015. Gila: Github label analyzer. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 479–483.

[8] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2014. The promises and perils of mining GitHub. In *Proceedings of the 11th working conference on mining software repositories*. ACM, 92–101.

[9] Riivo Kikas, Marlon Dumas, and Dietmar Pfahl. 2016. Using dynamic and contextual features to predict issue lifetime in GitHub projects. In *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 291–302.

[10] Antonio Lima, Luca Rossi, and Mirco Musolesi. 2014. Coding together at scale: GitHub as a collaborative social network. In *Eighth International AAAI Conference on Weblogs and Social Media*.

[11] Ehsan Noei, Feng Zhang, Shaohua Wang, and Ying Zou. 2019. Towards prioritizing user-related issue reports of mobile applications. *Empirical Software Engineering* 24, 4 (2019), 1964–1996.

[12] Raphael Pham, Leif Singer, Olga Liskin, Fernando Figueira Filho, and Kurt Schneider. 2013. Creating a shared understanding of testing culture on a social coding site. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 112–121.

[13] Erik Van Der Veen, Georgios Gousios, and Andy Zaidman. 2015. Automatically prioritizing pull requests. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 357–361.

[14] Bogdan Vasilescu, Vladimir Filkov, and Alexander Serebrenik. 2013. Stackoverflow and github: Associations between software development and crowdsourced knowledge. In *2013 International Conference on Social Computing*. IEEE, 188–195.

[15] Bogdan Vasilescu, Daryl Posnett, Baishakhi Ray, Mark GJ van den Brand, Alexander Serebrenik, Premkumar Devanbu, and Vladimir Filkov. 2015. Gender and tenure diversity in GitHub teams. In *Proceedings of the 33rd annual ACM conference on human factors in computing systems*. ACM, 3789–3798.

[16] Bogdan Vasilescu, Stef Van Schuylenburg, Jules Wulms, Alexander Serebrenik, and Mark GJ van den Brand. 2014. Continuous integration in a social-coding world: Empirical evidence from GitHub. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 401–405.

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