2015 12th Working Conference on Mining Software Repositories

Automatically Prioritizing Pull Requests

Erik van der Veen
Delft University of Technology
the Netherlands
Email: erikvdv1@gmail.com

Georgios Gousios
Radboud University Nijmegen
the Netherlands
Email: g.gousios@cs.ru.nl

Andy Zaidman
Delft University of Technology
the Netherlands
Email: a.e.zaidman@tudelft.nl

***Abstract*—In previous work, we observed that in the pull-based development model integrators face challenges with regard to prioritizing work in the face of multiple concurrent pull requests. We present the design and initial implementation of a prototype pull request prioritisation tool called PRioritizer. PRioritizer works like a priority inbox for pull requests, recommending the top pull requests the project owner should focus on. A preliminary user study showed that PRioritizer provides functionality that GitHub is currently lacking, even though users need more insight into how the priority ranking is established to make PRioritizer really useful.**

I. Introduction

Pull-based development as a distributed development model is a distinct way of collaborating in software development. In this model, the project's main repository is not shared among potential contributors; instead, contributors fork (clone) the repository and make their changes independent of each other. When a set of changes is ready to be submitted to the main repository, they create a pull request, which specifies a local branch to be merged with a branch in the main repository. A member of the project's core team (the *integrator*) is responsible to inspect the changes and integrate them into the project's main development line.

In earlier work [1], we surveyed 750 pull request integrators from high volume projects and discovered that the top two challenges they face when working with pull requests are maintaining project quality and prioritizing work in the face of multiple concurrent pull requests. With respect to pull request prioritization our findings are summarized in Figure 1.

In this paper, we present the design and initial implementation of a prototype pull request prioritization tool, the PRioritizer. PRioritizer works as a priority inbox for pull requests: it examines all open pull requests and presents project integrators the top pull requests that potentially need their immediate attention. It also offers an alternative view to Github's pull request interface, which allows developers to sort open pull requests on a multitude of criteria, ranging from the pull request's age to its number of conflicts with other open pull requests (pairwise conflicts). PRioritizer is a service-oriented architecture build on top of GHTorrent [2]: it uses GHTorrent's data collection mechanisms to react in near real-time to changes in pull request state.

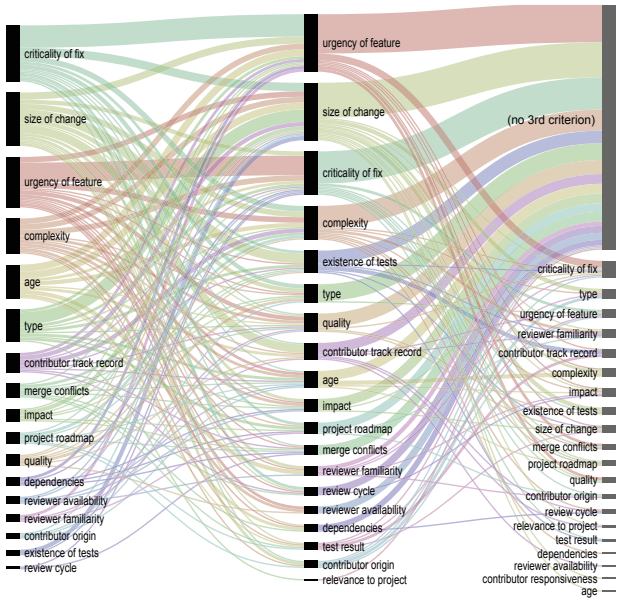

Fig. 1: Prioritization criteria and their order of application as reported by 750 integrators [1].

II. Prioritizing pull requests

A. Modeling

Contrary to work prioritization approaches that support development-oriented decisions (e.g. bug triaging) [3], [4], [5], we modelled the pull request prioritization using the priority inbox approach [6]. The difference lies in that we do not only look at static information with regard to the pull requests, but we also take into account (the dynamics of) previous actions on pull requests. What PRioritizer tries to do is present the integrators with the pull requests that will need their immediate attention.

To select the important pull requests, PRioritizer uses machine learning. Initially, time is split in configurable time windows (currently, 1 day long). In each time window, PRioritizer calculates a list of features for all pull requests (dependent variables, explained below) and a boolean response variable that indicates whether the pull request received a user action in the following time window. A machine learning algorithm is then trained on historical data to build a model for predicting whether current pull requests will receive user updates in the following time window. The pull requests with the highest probability to be updated are classified as the important ones.

978-0-7695-5594-2/15 $31.00 © 2015 IEEE
DOI 10.1109/MSR.2015.40

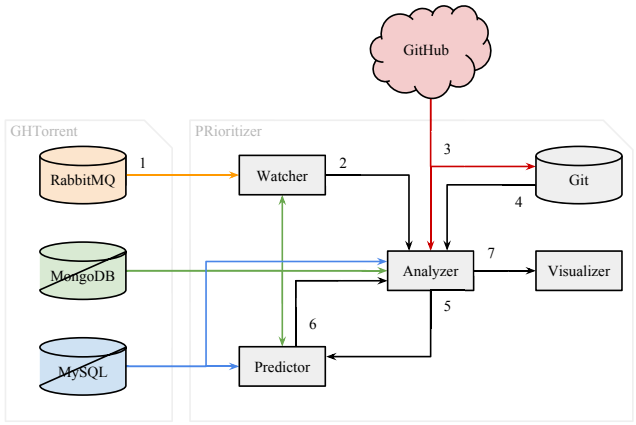

Fig. 2: Diagram of the architecture. It shows the different data sources and components used by the PRioritizer service.

B. Features

Our feature set was extracted by analysing the results of the survey [1] and closely correspond to the developer's answers as reported in Figure 1. An overview of the selected features can be seen in Table I.

One of the top parameters that developers examine when prioritizing is the *size of change*. We therefore include 4 related features in our model, namely additions, deletions, commits and files. Developers also deem the *age* of the pull request important: we measure it within the examined time window as the elapsed time between the time window start and the pull request creation. The *contributor's track record* is also taken into account by integrators. To approximate the track record, we use 3 features: whether contributors are core team members, their contribution rate and their pull request acceptance rate. The contribution rate is calculated by taking the number of commits which are authored by the contributor and included in the project divided by the total number of commits in the project. Finally, a feature that emerges as important second prioritization criterion is the *existence of tests*. In some projects it is often expected that new pull requests contain tests for the code they modify or add. Tests are identified by simple heuristics: if "test" or "spec" are in the file path of any file affected by the pull request, then the pull request is marked as having tests.

III. Design and Implementation

We designed PRioritizer as a loosely-coupled architecture based on independent micro services. Figure 2 shows a global overview of the architecture. The PRioritizer service uses two main data sources: GitHub and the GHTorrent project. When an event arrives the *watcher* component is notified (1) and starts prioritizing the project (2). When the *analyzer* gets a prioritization request, it fetches a list of open pull requests from GitHub and the pull request contents to the local Git clone (3). When the data is fetched, the analyzer processes each pull request in the list with data from the local clone and GHTorrent (4). The data is now ready to be processed by the *predictor* (5), which generates an ordering for the pull requests. After the ordered list is returned to the analyzer (6), the output is generated and available for the *visualizer* (7). Details about the design are presented in the following sections.

a) GHTorrent: GHTorrent [2] mirrors all data exposed from the GitHub API in real time. It monitors the GitHub event timeline API endpoint and publishes the retrieved data to a queue service (RabbitMQ) where multiple clients can connect to. It also maintains 2 databases, an unstructured one (MongoDB) which contains the raw replies from the GitHub API and a relational one (MySQL) which stores indexed historical data for all GitHub repositories. PRioritizer uses GHTorrent as a source for both live and historical data. The live data that the PRioritizer is interested in are events on pull requests triggered by actions such as assignment of the pull request to a specific user or merging the pull request.

b) Watcher: The watcher listens to pull request events from GHTorrent via a RabbitMQ message queue. It maintains a list of registered repositories and informs the analyzer when pull request events for one of those is repositories is received.

c) Analyzer: The analyzer analyzes pull request events and computes values for the features presented in Section II-B. To do so, for each repository, it maintains a local Git checkout and a list of open pull requests. On every pull request event, it updates both the Git repository with a branch corresponding to the source branch of the pull request and the open pull request list with fresh information from GHTorrent. Then, it uses both GHTorrent and information into the raw pull request data to calculate all features in the current time window.

The analyzer also implements a high-performance pull request pairwise conflict detection mechanism that works by simulating pairwise merges in memory. To avoid recomputing valid branch merges, it caches intermediate results.

To maintain high performance, all independent processes (Git repository updating, conflict detection etc) are initiated asynchronously and their results are gradually composed towards a final set of metrics for the processed pull request.

d) Predictor: The predictor calculates the probability that a specific pull request will be active within the next time window. To do so, it maintains a per repository model extracted by applying a machine learning algorithm to existing pull requests and then uses the computed model to calculate probabilities for currently updated pull requests.

The predictor is split in two parts: the historic data calculator and the machine learning implementation. The first shares code (but not the runtime) with the analyzer as both tools basically compute the same values in different time windows. To capitalize on the wealth of available options in statistical environments such as R, the machine learning part is implemented as a different service that communicates with the main predictor process through file exchange.

e) Visualizer: The visualizer visualizes a list of pull requests, enriched with information extracted from the analysis and prediction phases. To address reported deficiencies in GitHub's pull request user interface (also discovered through our user survey), the visualizer supports filtering pull requests

Feature	Description	5%	Mean	Median	95%	Plot
Age	Minutes between open and the current time window start time.	0.00	167344.02	77760.00	646560.00	
Contribution Rate	The percentage of commits by the author currently in the project.	0.00	0.03	0.00	0.094	
Accept Rate	The percentage of the author's other PRs that have been merged.	0.00	0.45	0.50	0.90	
Additions	Number of lines added.	1.00	3649.86	41.00	6285.00	
Deletions	Number of lines deleted.	0.00	2271.32	7.00	2353.00	
Commits	Number of commits.	1.00	6.52	2.00	22.00	
Files	Number of files touched.	1.00	53.88	2.00	125.00	
Comments	Number of discussion comments.	0.00	4.22	1.00	17.00	
Review Comments	Number of code review comments.	0.00	1.60	0.00	8.00	
Core Member	Is the author a project member?	0.00	0.26	0.00	1.00	
Intra-Branch	Are the source and target repositories the same?	0.00	0.06	0.00	1.00	
Contains Fix	Is the pull request an issue fix?	0.00	0.098	0.00	1.00	
Last Comment Mention	Does the last comment contain a user mention?	0.00	0.091	0.00	1.00	
Has Test Code	Are tests included?	0.00	0.35	0.00	1.00	

TABLE I: Selected features and descriptive statistics for predicting pull request activity. The calculation unit is a pull request. Histograms (red) are in log scale.

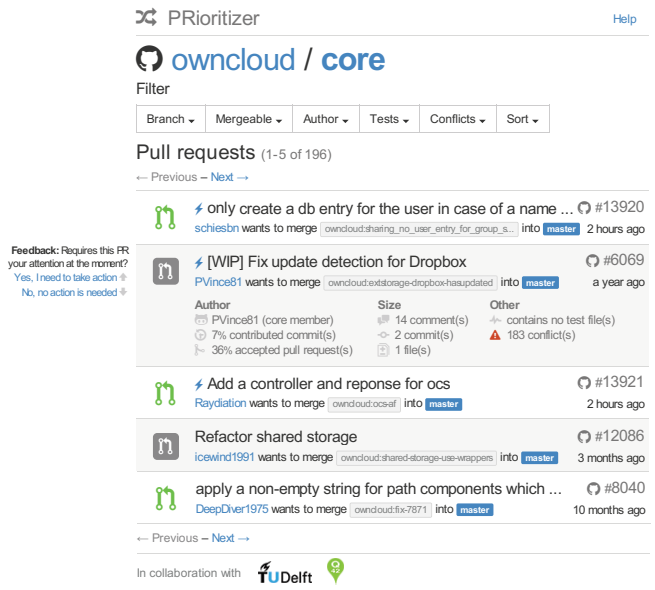

Fig. 3: The user interface shows an ordered list of pull requests that need attention.

based on specific branches, originating in specific authors, having conflicts or tests and being in a mergable state. It also supports sorting pull requests based on a multitude of criteria, including all features reported in Section II-B. A key feature of the visualizer is support for user feedback on the prioritization; the user can report if the proposed prioritization order is correct and change it to indicate what is the correct one. At the moment, this information is only collected; in the future, we plan to use it in order to improve the prioritization.

The visualizer is implemented as a static web application; a screenshot of its main page can be seen in Figure 3.

IV. Initial Evaluation

A. How good are the activity predictions?

To select an algorithm to base our prediction engine on, we used historical data from 475 projects and three commonly used machine learning algorithms: Logistic Regression, Naïve Bayes and Random Forests. The target of the test was to compare how well models build with each algorithm could predict whether a pull request would become active in the next time window against the ground truth (what actually happened). We ran the three algorithms against each project with a 10-fold random selection cross-validation. The models were trained with 90% training data and 10% testing data.

The results show that both Logistic Regression and Naïve Bayes perform badly on precision (0.36 and 0.34) and accuracy (0.62 and 0.60), which were our top priorities. On the other hand, Random Forests perform better overall with a precision of 0.64 and an accuracy of 0.85. To improve the performance, we tried several optimisations like dataset balancing and feature pruning to no avail.

The results show that using Random Forests, we can predict with relatively high accuracy (86% on average across projects) whether a pull request in a given time frame of its life will be active in the next one. This is an important result for building a prioritizer service, as it gives us the confidence to recommend a default ordering to the service user. Moreover, as the quality of the prediction varies across projects, we can build preprocessing phases to determine whether default recommendations can be useful. However, there is room for improvement: by selecting more representative features, or custom features for each project, we can account for variations of pull request handling practices. Moreover, user directed evaluation (already implemented in the visualizer) can help retrofit our machine learning model with user preferences.

B. Performance characteristics

The PRioritizer runs incrementally on a set of pull requests. An initial import involving training on a repository takes significantly longer than a normal run. The time required to import ranges from a few minutes, for projects with less than a hundred pull requests to a few hours for the biggest of projects. After initial import, the prioritization time depends on the number of open pull requests: for a medium sized project (30 open pull requests), the processing takes less than a second, As a service, the PRioritizer is run on a machine featuring a 3 GHz Quad core CPU and 16 GB of RAM. Without significant effort

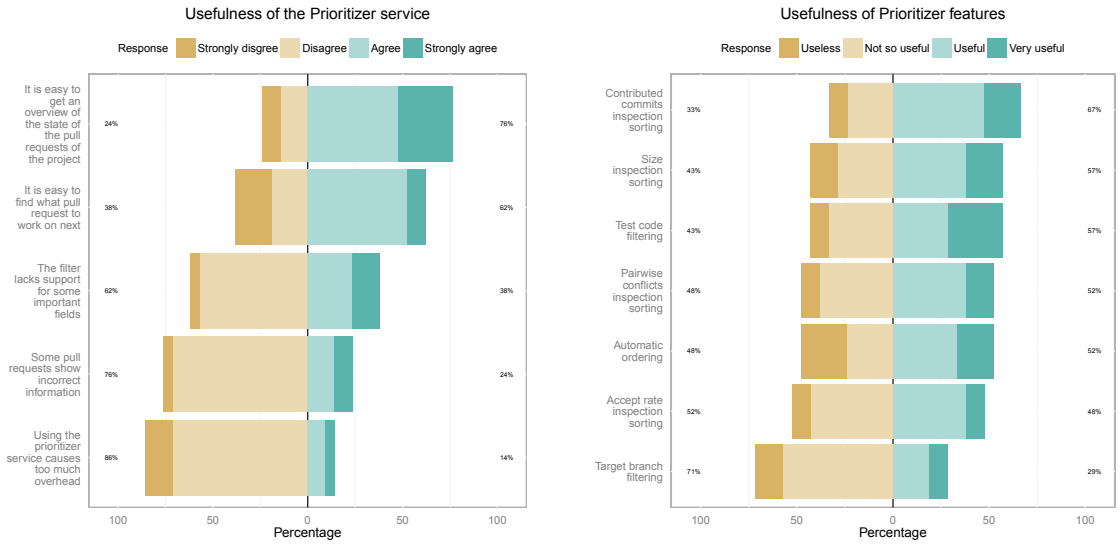

Fig. 4: User evaluation of the service and individual feature usefulness

put on performance optimization, it is capable of prioritizing 7 pull requests and can simulate 400 pairwise merges per second.

C. User Evaluation

We performed a preliminary user evaluation in order to guide our future developments of PRioritizer. For each of the 450 projects we used for model building, we invited core team members of those projects to use it. The invitation was sent by personal email containing a private link to the prioritizer installation for their repository and a survey. The survey consisted of 4 odd Likert-scale questions and 4 open ended ones. The Likert-scale questions invited users to rate the usefulness of specific features and the overall service quality while the open ended questions asked the users for missing features or potential improvements. We received 21 answers.

Overall, as can be seen in Figure 4, the usability of PRioritizer has been positively perceived by human evaluators. They overwhelmingly state that PRioritizer would not cause extra overhead (86%) and that the PRioritizer interface can provide an easy to comprehend overview of the overall project pull request status (76%). In the open ended responses, the users indicated that pairwise conflict detection, target branches and the pull request contributor profile were favourite features. As R1 states: "*I can see at a glance which PRs can be merged automatically. For some reason the Github PR interface does not show this, you have to click on a PR to find out if it can be automatically merged...*".

The main highlight of the service, automatic prioritization, received mixed reviews; an equal number of people think that it is very useful and not useful at the same time. From the open ended answers, we found that integrators do not understand how automatic prioritization works. As R17 puts it: "*It can show us the most pressing pull requests. However, it is unclear how this ranking is established, so I'd hoped to know why a pull request is considered more urgent then others*". As such, 86% of the integrators mentioned that they want more insight on how the automatic ordering works in future versions.

Finally we asked the respondents if they would use PRioritizer for their project. 57% (12) of the respondent gave a positive answer. We correlated the size of the project to the probability that integrators would give a positive answer to the above question and found that integrators in bigger projects tend to find prioritizer more useful ($\rho = 0.4307$).

V. Conclusions and Future Work

In this paper we have presented a pull-request prioritization approach called PRioritizer. PRioritizer enables a priority inbox style approach for integrators that face challenges in the face of multiple concurrent pull requests. The priority inbox approach does not only take static information with regard to pull requests into account, instead it also considers the dynamics of pull requests. A preliminary user study has shown that PRioritizer provides functionality that GitHub is currently lacking. In particular, the participants appreciated the easy to comprehend overview of the overall project pull request status, the pairwise conflict detection, target branches and pull request contributed profile. The verdict for the automatic prioritization is mixed, with participants requesting more insight on how the automatic ordering works.

Our three main avenues for future work are: (1) adding more insight into how the prioritization works, (2) enabling the incorporation of the user feedback into our algorithm and (3) improving scalability.

Acknowledgements

The authors would like to thank Audris Mockus for discussions that influenced the design of the prioritization algorithm.

REFERENCES

[1] G. Gousios, A. Zaidman, M.-A. Storey, and A. van Deursen, "Work practices and challenges in pull-based development: The integrator's perspective," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015, to appear.

[2] G. Gousios, "The GHTorrent dataset and tool suite," in *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)*. IEEE Press, 2013, pp. 233–236.

[3] J. Anvik and G. C. Murphy, "Reducing the effort of bug report triage: Recommenders for development-oriented decisions," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 10:1–10:35, Aug. 2011.

[4] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*. ACM, 2009, pp. 111–120.

[5] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Fuzzy set and cache-based approach for bug triaging," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2011, pp. 365–375.

[6] D. Conway and J. M. White, *Machine Learning for Email: Spam Filtering and Priority Inbox*. O'Reilly, 2011.