KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Computer Science and Engineering

Second cycle, 30 credits

The temporal side of pull request acceptance

BALTHAZAR WEST

Stockholm, Sweden 2022

The temporal side of pull request acceptance

BALTHAZAR WEST

Master in Computer Science
Date: December 3, 2022
Supervisor: Fernanda Madeiral
Examiner: Martin Monperrus
School of Electrical Engineering and Computer Science
Swedish title: Den tidskänsliga sidan av pull request-acceptans

Abstract

The contemporary way of contributing to open-source software is through online platforms. GitHub is the most widely used platform for this purpose. On GitHub, users can suggest improvements to projects by opening a *pull request* (PR), taking on the role of the *submitter*. The PR is then judged by a *reviewer* who can ask for adjustments before the PR is closed and thus merged into the project or rejected. Several tools have been created that recommend PRs for reviewers to increase productivity. The models powering such recommendations are based on closed PRs, yet they make predictions on open PRs. The relative performance change between making predictions on open and closed PRs remains unknown. Furthermore, surveys have been conducted to gain an understanding of PR usage, however, they have missed key moments in a PR's lifecycle. Insights into these aspects could boost productivity by improving recommendation tools and increase the merge rate by enhancing the development and review process of users.

This thesis presents our mixed-methods study to address these concerns. We modeled and compared the performance of predicting the merging outcome of open and closed PRs. Predicting the outcome of open PRs had significant drawbacks, particularly non-merge predictions. However, merge predictions were satisfactory and all predictions improved exponentially as PRs neared completion. Then, we measured the importance of each factor on open and closed PRs. We discovered that the top factors for closed PRs were also the most important for open PRs, but occasionally in the reverse direction, i.e., they hurt predictions. We also found two factors that could be modeled with no downside and occasional performance gain. Moreover, we surveyed GitHub users to find out how submitters prepare PRs and respond to feedback and how reviewers judge and respond to PRs. The submitters shared that they prepare PRs by emphasizing project fitness, relevance, and clarity, and they try to respond quickly by fixing flaws and clarifying their intent. The reviewers shared that they always value project fitness, they appreciate simplicity and clarity in the first PR review, and they respond by asking for code improvements and more compliance with conventions. Our findings can enhance the usage of PRs by improving recommendations and collaboration processes.

Keywords

Pull requests, GitHub, code review, machine learning, survey

Sammanfattning

Det moderna sättet att bidra till mjukvara som är open source är via onlineplattformar. GitHub är den mest populära plattformen för detta syfte. På GitHub kan användare föreslå förbättringar till projekt genom att öppna en *pull request* (PR) och därmed anta rollen som *sändare*. PR:n är sedan granskad av en *recensent* som kan be om justeringar innan PR:n stängs och därmed blir accepterad eller nekad. Flertal verktyg har skapats som rekommenderar PR:s till recensenter för att öka produktivitet. Modellerna som verkställer dessa rekommendationer är baserade på stängda PR:s trots att de gör förutsägelser på öppna PR:s. Prestandaskillnaden mellan att göra förutsägelser på öppna och stängda PR:s har inte utforskats. Vidare har enkäter genomförts för att bättre förstå användningen av PR:s, dock har de missat viktiga moment i livscykeln av en PR. Insikter inom dessa aspekter kan öka produktivitet genom att förbättra rekommendationsverktyg och öka framgången av PR:s genom att förbättra processen av utveckling och recension.

Vi presenterar vår tes av mixade metoder för att angripa dessa punkter. Vi modellerade och jämförde prestandan av att förutspå beslutet av öppna och stängda PR:s. Att förutspå beslutet av öppna PR:s hade signifikanta nackdelar, specifikt förutsägelser av nekade PR:s. Dock var förutsägelsen av accepterade PR:s duglig och prestandan ökade exponentiellt när PR:s närmade deras slutgiltiga tillstånd. Efteråt mätte vi inflytandet av varje faktor på öppna och stängda PR:s. Vi upptäckte att toppfaktorerna för stängda PR:s var även viktigast för öppna PR:s, men ibland i motsatt riktning, alltså skadades prestandan. Vi fann också två faktorer som kunde modelleras utan nackdelar och med enstaka fördelar. Vidare skapade vi en enkät på GitHub-användare för att se hur sändare förbereder PR:s och svarar på feedback och hur recensenter granskar och svarar på PR:s. Sändarna sa att de förbereder PR:s genom att betona projektanpassning, relevans, och tydlighet, och de försöker svara snabbt genom att fixa problem och förtydliga deras avsikt. Recensenterna sa att de alltid värderar projektanpassning, uppskattar simplicitet och tydlighet i första PR-recensionen, och de svarar genom att be om kodförbättringar och mer följsamhet med konventioner. Våra upptäckter kan förhöja användandet av PR:s genom att förbättra rekommendationer och samarbetsprocesser.

Nyckelord

Pull requests, GitHub, kodrecension, maskininlärning, enkät

Acknowledgments

I would like to thank my supervisor Fernanda, for without her willingness to be ambitious, my thesis would not have unlocked its potential. I also want to thank my examiner Martin for letting me divert from his original idea and work on what I found most valuable. The people in my personal life deserve recognition as well. Thank you to Gustav, Marcus, Oskar, and Rickard, for without them, I would never have been in a position to finish my master's degree. I am also deeply grateful for the support from my long-lasting friends and family, especially my mother.

Stockholm, December 2022
Balthazar West

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Open-source software projects are commonly stored in central repositories. Nearly all of these projects use a distributed version control system. The key aspect of such systems is that people can suggest specific changes to incrementally improve a project that can be merged smoothly. Team members and external users alike may take on the *submitter* role by developing and submitting suggested changes. *Reviewers* are team members that act as gatekeepers, tasked with merging or rejecting suggestions. Alternatively, they can provide feedback if an unsatisfactory suggestion can become acceptable after some modification. It is the submitter's responsibility then to address the feedback to reach approval. This process is known as pull-based development, which allows collaboration and code review.

Several platforms host repositories with their respective histories and provide mechanisms for pull-based development. GitHub is the most widely used, hosting millions of open-source projects under a distributed version control system. On GitHub, suggested changes are packaged in *pull requests* (shortened to PR from hereon) for reviewers to inspect. As repositories mature, the volume of PRs increases while the amount of active reviewers remains stable [1]. Naturally, this increases precious time spent by reviewers rejecting PRs, either outright or after potentially several rounds of feedback, while submitters face more rejection. Reviewers have expressed being overwhelmed with pending PRs [2] and submitters have expressed discouragement after being rejected [3]. To decrease wasted effort and discouragement, both parties need insights as to how they should cooperate to develop PRs for successful integration and avoid fruitlessly discussing those destined for rejection.

1.1 Problem statement

Plenty of work has been done related to PRs on GitHub. Many models have been created to predict the merging outcome of a PR and the importance of each factor has been measured [4, 5]. Other studies have opted to model the lifetime of a PR instead and find the factors that influence it [6, 7]. In an attempt to increase productivity and alleviate workload, tools that recommend PRs for review have been implemented, deployed, and evaluated with good outcomes [8, 9]. Also, a few surveys have been conducted to understand how GitHub users develop and review PRs [2, 10]. While these studies have brought valuable findings, the vast majority of them have not emphasized the temporal dimension of PRs.

First, it is important to realize that previous models have performed predictions on both open and closed PRs, i.e., PRs before and after being accepted or rejected. While existing studies have reported the quality of their predictions, none of them have compared the quality difference between predicting on open and closed PRs. Insights on this could help developers of PR recommendation tools estimate how the performance of their tool changes as open PRs approach completion. Moreover, many studies have measured the importance of each factor in their model. However, none of them have compared how the importance of each factor changes between predicting on open and closed PRs. This could guide tool developers on what they should focus on when creating their underlying models.

Second, while surveys have been conducted on how GitHub users develop and review PRs, some aspects of their usage have yet to be explored. Specifically, submitters have not been asked how they respond to feedback on their PRs. Even though reviewers have shared how they judge submissions, they have not been asked specifically in the context of the first and final review of a PR. Also, the surveys have not covered the interplay between submitters and reviewers across the entire lifecycle of a PR, from its preparation to its final decision. Learning how PRs are developed and reviewed could benefit GitHub users in various ways, e.g., novices could learn useful pointers and guidelines for PR submission that can be improved. The benefits would increase the potency of PRs, decrease wasted effort, and alleviate the aforementioned feelings of frustration.

1.2 Goals

The overarching goal of this work is *to investigate the temporal side of pull request acceptance*. To serve this goal, we elaborated the following research questions:

RQ1: How does the prediction of acceptance change when predicting on open pull requests? The goal is to investigate whether the acceptance of open PRs can be predicted based on data from closed PRs. In particular, we want to examine the predictions of models trained on closed PRs and tested on PRs in various states. To do so, we trained and evaluated machine learning classifiers using PR data from three points in their timeline.

RQ2: How important is each factor when predicting the acceptance of pull requests across their lifetime? The goal is to gauge the importance of each factor on acceptance at multiple points of a PR's timeline. We measured the importance of factors by retraining and reevaluating our most accurate model without the factor of interest. We repeated this step for every factor and every PR state.

RQ3: How do submitters prepare pull requests and respond to feedback to increase acceptance? The goal is to find out how submitters develop PRs intending to increase acceptance. To address this, we surveyed PR submitters about how they prepare their PRs and respond to feedback.

RQ4: How do reviewers judge and respond to pull requests to increase acceptance? The goal is to find out how reviewers conduct code reviews intending to increase acceptance. To address this, we surveyed PR reviewers about how they judge and respond to PRs.

1.3 Methods

We answered RQ1 and RQ2 quantitatively by modeling the acceptance of PRs on GitHub. First, we selected 100,000 recent PRs from 239 repositories in the top programming languages. We then built the timeline of each PR, i.e., their sequence of events from opening to closing. The timelines were used to recreate each PR in the following three states: when it was ready for review, halfway complete, and complete. We extracted 19 factors from each PR state, of which five factors were novel. Then, we trained seven machine learning

classifiers on the data set of closed PRs using 10-fold cross-validation. Our models predicted the acceptance of the PRs in each state and we reported the prediction quality to answer RQ1. Our most accurate model was then used to measure factor importance. The drop-column strategy was used, i.e., we retrained and reevaluated our model without each factor and reported the accuracy change to quantify importance. We repeated this measurement for each factor and state and reported them to answer RQ2.

We addressed RQ3 and RQ4 qualitatively with an open survey on GitHub users. We created a questionnaire for both submitters and reviewers. The section for submitters asked how they prepare PRs and respond to feedback, and the section for reviewers asked how they respond to PRs and judge them in the first and final review. These questions were multiple-choice with free-text options and they were explicitly asked in the context of increasing acceptance. Our questionnaire concluded with compulsory demographic questions. Several parties gave feedback on the questionnaire before public release. We invited two groups via email: 2,000 GitHub users who had shown recent activity on PRs and 68 authors of related works. Our invitations were personalized, described the context, completion time, and anonymity, and encouraged sharing the questionnaire. We received 204 responses, which were filtered, presented, and summarized to answer both research questions.

1.4 Contributions

The primary contributions of this thesis are the following:

- A quantitative study on GitHub data to gauge if closed PRs can be used to predict the acceptance of open PRs and to measure the importance of each factor in such predictions.
- A qualitative study on GitHub submitters and reviewers to learn how accepted PRs are developed and reviewed from their preparation to the final decision.

Our quantitative study produced mixed results. While our merge predictions of open PRs were successful, non-merge predictions suffered greatly, occasionally performing worse than naive models. However, all metrics improved exponentially when tested against PRs nearer their final state. The top factors for closed PRs were the number of approved code reviews, last CI build status, and amount of discussion. These were also highly important in the

open states, but sometimes in the reverse direction, meaning that their effect was particularly sensitive to the current state of the PR.

With the qualitative study, we found that submitters most commonly prepared their PRs by emphasizing project fitness, relevance, and clarity, respectively. They typically responded to feedback by trying to respond quickly, fixing flaws, and clarifying their intent. As to the reviewers, they appreciated simplicity and clarity in the first review of a PR, while project fitness always remained a top priority. They typically asked for code improvements and more compliance with conventions.

Other contributions were made as a result of planning and executing our studies. We consider these secondary contributions and they are highlighted as follows:

- A review of the pull request factors from the state of the art (Table 2.1).
- A novel method for collecting temporal pull request data (Section 3.2.2).
- Implementations for collecting and modeling temporal pull request data.
- A data set of temporal data from 100,000 pull requests.
- A survey design on GitHub users with 204 responses.

The last three items are available online[1] for the sake of open science and usage in future works. All of the supplementary material referenced throughout this thesis is available at the link.

1.5 Structure of the thesis

This thesis is structured in the following way. Chapter 2 provides background on modern software development and presents the state of the art on understanding pull request acceptance, lifetime, recommendation, and user actions and opinions. Chapter 3 presents the design of this study, including how machine learning was applied and how a survey was conducted to answer our research questions. In Chapter 4, we report the results obtained by the execution of the study, and we discuss the implications of the results and contrast them against the related works in Chapter 5. Finally, in Chapter 6, we conclude the work and present avenues for future research.

[1] https://github.com/Wesbalt/state-of-the-pr.

Chapter 2

Background & State of the art

In this chapter, we provide the background on concepts and terminology necessary to understand this thesis. Specifically, we describe modern software development (Section 2.1), including pull-based development and GitHub, and we review the related works (Section 2.2).

2.1 Modern software development

This section lays out common aspects of modern software development. We describe version control systems in Section 2.1.1 and the pull-based development model in Section 2.1.2. Moreover, we introduce the coding platform GitHub and some of its features necessary to understand the rest of this thesis in Section 2.1.3.

2.1.1 Version control systems

As a software project matures over time, tracking file versions becomes a near necessity. This practice is called *version control*. Approaching this manually is both cumbersome and prone to mistakes, so developers often use a *version control system* to facilitate version control automatically. Such systems store a central *repository*, i.e., the project files and the history of file changes, from which developers may download a local copy called a *clone*. Developers with permission can *push* their local changes, e.g., a bug fix, to this repository and the rest of the team can then *pull* the changes into their clones. This way, everyone maintains a synchronized copy of the project.

Larger teams can benefit from working on independent streams of changes in the same project that may be integrated into the central repository once

finished. Version control systems organize this workflow in *branches*, where each branch is typically assigned one feature implementation or improvement. Upon finishing development in a branch, it is time to integrate the changes into the central repository, called a *merge*. If a *conflict* is detected, e.g., if there are competing line changes in the same file, the user is tasked with resolving it before merging. Users can check out any branch at any time and return to the "un-branched" codebase, called *master* or *trunk*. Furthermore, they can compare file changes over time and swap between different repository versions. The latter is especially useful for debugging since users can trace when a particular bug was introduced.

There are multiple version control systems available, including Git, Apache Subversion, and Team Foundation Version Control[1].

2.1.2 Pull-based development

Pull-based development is a model of collaborative development using a version control system where project changes are suggested through *pull requests* (PRs). There are two key roles: the *submitter*, i.e., the person who develops and submits the PR, and the *reviewer*, i.e., the person who judges and decides to accept or reject the PR. The submitter may be a team member, or an external user if allowed, and the reviewer is typically a team member who is familiar with the codebase.

Figure 2.1 shows the interplay between the submitter and reviewer. First, the submitter either creates a personal clone of the target repository, called a *fork*, or branches it. Then, they make the desired changes and submit a PR. The reviewer is tasked with judging the PR by assessing quality, relevance, and much more as evidenced in Section 2.2. This assessment is called *code review*. If the PR is deemed unsatisfactory, the reviewer can give feedback which the submitter can respond to, and several rounds of this may be necessary. Ultimately, the reviewer decides to either accept the PR, which means merging the changes into the repository, or reject it. Either way, the PR closes and both parties can move on to another PR.

Plenty of websites and desktop applications support pull-based development, including GitHub, GitLab, Bitbucket, and Azure DevOps Services[2].

[1] Available at `https://git-scm.com`, `https://subversion.apache.org`, `https://docs.microsoft.com/en-us/azure/devops/repos/tfvc`. Last access 2022-11-30.
[2] Available at `https://github.com`, `https://gitlab.com`, `https://bitbucket.org`, `https://azure.microsoft.com/en-us/services/devops`. Last access 2022-11-30.

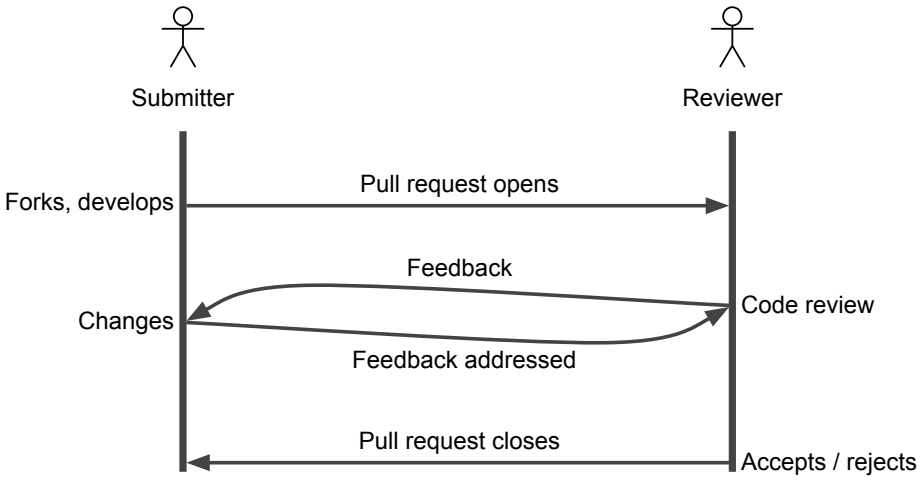

Figure 2.1 – Interplay between the submitter and reviewer on a pull request.

2.1.3 GitHub

GitHub is a cloud-based Git repository hosting service available online, on their desktop GUI, or through the command-line interface *git*. Both the website and GUI also support pull-based development, but the website has additional social, management, and repository features. It is the most popular source code host, boasting over 100 million users and more than 360 million repositories[3]. This subsection lays out the commonly used features necessary to understand this thesis.

Repositories on GitHub

The repository is perhaps the most basic element of GitHub[4]. A repository may be viewed as a folder containing the project files and metadata files necessary to facilitate version control system operations. Repositories are augmented with management options, a wiki page, a statistics page, and a convenient button for users to fork repositories smoothly.

[3]Source: `https://github.com/search`. Last access 2022-11-30.
[4]Source: `https://docs.github.com/en/github/creating-cloning-and-archiving-repositories/creating-a-repository-on-github/about-repositories`. Last access 2022-11-30.

Commits on GitHub

Changes to a repository must be done by pushing *commits*. A commit is a grouping of changes to specific lines and entire files[5]. Every commit contains its changes, a message, timestamps, and the author. They are atomic by convention, meaning they are usually small and address a single thing, such as fixing one bug, adding one function, or cleaning up one file.

Issues on GitHub

Issues are used to track todo items, bugs, and feature requests in GitHub repositories[6]. They serve as a channel of communication between users where problems and features can be shared and discussed, team members and external users alike. A specialized tab allows users with permission to browse issues with a searchable and filterable list. Issues may be tagged using *labels* to quickly communicate the kind of issue at a glance, e.g., "bug" or "missing feature". Issues may also be assigned to users for coordination purposes.

Pull requests on GitHub

A PR contains a set of commits and is used by team members and external developers to submit changes to a branch[7]. As with GitHub issues, PRs have a title, description, and labels for communicating the changes. PRs are published under a special tab where they are subject to code review. If there is a conflict or the reviewer is not satisfied with the changes, they can close the PR immediately or give feedback and allow the submitter to revise the PR. If the PR becomes satisfactory, its changes may be merged into the codebase and the PR is closed. PRs may automatically close issues on merge by including special keywords in the title, e.g., "fix #37" closes issue 37.

Continuous Integration on GitHub

The code review process on GitHub may be enriched with *Continuous Integration* (CI). CI services work by monitoring the repository for new or updated

[5]Source: `https://docs.github.com/en/github/committing-changes-to-your-project/creating-and-editing-commits/about-commits`. Last access 2022-11-30.
[6]Source: `https://docs.github.com/en/issues/tracking-your-work-with-issues/about-issues`. Last access 2022-11-30.
[7]Source: `https://docs.github.com/en/github/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests`. Last access 2022-11-30.

PRs, running a task when they arrive, and reporting the outcome in the PR thread. For example, Travis-CI[8] is a popular CI service that performs automated testing by applying the changes locally to the branch the PR is trying to merge into, running a test suite, and reporting the outcome with a detailed error message if a failure occurred. Such services are helpful to both submitters and reviewers because they save time and catch errors early.

2.2 Research on pull requests

In this section, we review the related work on GitHub PRs. Specifically, we highlight the contributions to the understanding of the merge decision (Section 2.2.1), the lifetime of PRs (Section 2.2.2), the recommendation of PRs (Section 2.2.3), and the actions and opinions of PR users (Section 2.2.4). Lastly, we map the existing studies to the factors used to model PRs and for what purpose (Section 2.2.5).

The literature uses inconsistent language when referring to the submitter and reviewer. For consistency, we use the aforementioned terms unless they change the intended meaning, at which point we fall back to the original terms in the citation.

2.2.1 Understanding pull request acceptance

Many studies have been conducted to understand the merge decision, i.e., the acceptance of, PRs on GitHub. To this end, acceptance has been modeled numerous times [4, 5, 11] and PRs have been manually inspected [12, 13]. This section summarizes well-known studies, studies that add to previous work, and studies with novel contributions.

Gousios et al. [4] made several important contributions related to pull-based development. Specifically to the topic at hand, they modeled the acceptance of PRs using six different machine learning algorithms and 15 PR factors based on previous works. Notably, project-specific factors such as team size were considered. The study covered ∼170K PRs from hundreds of GitHub projects written in multiple programming languages. Then, they used their most accurate model to measure the importance of each factor. The merge decision was found to be influenced mainly by whether the PR modifies recently modified code. In fact, this factor, along with the size of the project's source

[8] Available at `https://github.com/travis-ci/travis-ci`. Last access 2022-11-30.

code and the number of files touched by the PR, could predict acceptance with sufficient accuracy.

Acknowledging that previous works have suggested a social structure surrounding contribution in open-source software, Tsay et al. [14] sought to study how both social and technical factors affect PR acceptance. The factors were categorized into three levels: the PR level, submitter level, and repository level. A statistical model was used to associate the factors with the merging decision. It was discovered that both technical and social factors were relevant for acceptance. Specifically, on the PR level, shorter social distance followed by shorter discussion increased acceptance, on the submitter level, higher social status increased acceptance, and on the repository level, more repository maturity decreased acceptance.

Kononenko et al. [6] performed a case study on a commercial project in which they modeled acceptance and shared the significance of each factor. Their Logistic Regression classifier considered the submitter's affiliation to be most important by far, where being affiliated with or close to the project owners increased your chances. The next strongest effect was the negative effect of lengthy discussions specifically by developers, however, more inline code comments by developers had the opposite and moderate effect. Finally, more submitter experience and larger PRs were associated with increased and decreased acceptance, respectively.

Dey and Mockus [15] added to the topic in a case study on the NPM ecosystem. Their data set was extensive, consisting of $\sim$500K PRs from $\sim$4K packages. First, they partially replicated another study on PR recommendation [8] by modeling the same selection of factors and found that it could successfully predict acceptance as well. They contrasted this against modeling their own factor selection, including original factors related to the submitter's and repository's history, which yielded a significant improvement. Lastly, the authors predicted the acceptance of newly submitted PRs. Their rationale was that models are usually trained and tested on closed PRs, and this does not reflect the use case of recommendation tools that must predict on open PRs. By removing three time-sensitive factors and retraining, they found that predicting open PRs decreased performance.

The authors of the aforementioned study continued their investigation on NPM packages in another study [16] using similar data and factors. Both the first [15] and second [16] study addressed the limitations of previous works by exploring whether there is a "sweet spot" for each factor to maximize acceptance, rather than showing a linear relationship. We summarize only the findings of this topic for the second study as it presented more thorough find-

ings. They discovered a non-linear relationship between factors and acceptance. For instance, acceptance peaked for PRs with 20 lines added and it dropped steadily up to 400 lines, at which point it kept dropping but slower. The authors also found that technical, social, and repository factors were all significant for PR acceptance, as suggested by Tsay et al [14]. In terms of importance, the age of the PR was dominant for the merge decision. The repository characteristics were second most important, followed by the amount of discussion, PR size, and submitter experience.

A few studies have focused on CI in relation to acceptance, with Yu et al. [5] being one of the first. They measured the effect of CI factors and how they affect the already established PR factors. The data set was broad, covering hundreds of projects written in multiple programming languages that all used the Travis-CI service. When replicating a previous model, they found older projects to have more acceptance, conflicting with the finding of Tsay et al. [14]. Yu et al. rebuilt the model with two additional CI-related factors and found that they factors were significant but did not majorly offset the importance of the other factors. In particular, PRs that failed the CI check were $\sim$90% more likely to be rejected. Zampetti et al. [17] studied CI factors further and reached a similar verdict. They trained multiple machine learners on GitHub projects that used Travis-CI and four CI factors were used. While they found that CI-related factors were ranked high by the predictors, previously established factors still dominated.

Two notable studies have discussed code quality in relation to the acceptance of PRs. Zou et al. [7] performed an exploratory study on GitHub Java projects to see how code style consistency affects the merge decision. Inconsistency was measured by the difference between the files touched by the PR and the original files. They based the measurement on 37 code style metrics based on broadly accepted standards and used eight confounding factors when modeling. Overall, inconsistency had a small negative effect on the merge decision, meaning that large inconsistency correlated with rejection. Considering each kind of code style difference as a separate factor, inconsistency with maximum line lengths and usage of tabs had the largest effect, six factors were insignificant, and the rest were negligible.

Lenarduzzi et al. [12] researched the influence of code quality on acceptance. PMD[9], a cross-language static code analyzer, was used to identify code quality issues such as code smells, security vulnerabilities, and code style violations. The authors first sought to find the prevalence of code quality issues. 95% of the PMD rules were violated by both accepted and rejected PRs, the

[9]Available at `https://pmd.github.io/`. Last access 2022-11-30.

vast majority of which had a violation of "medium" priority. Next, they studied the influence of PMD issues on acceptance. Applying logistic regression and six machine learning techniques, they did not find a significant difference between accepted and rejected PRs w.r.t. PMD issues. A sample of PRs was manually inspected to complement these results. None of them were rejected due to code quality issues, leading to the perhaps counter-intuitive conclusion that code quality issues did not affect acceptance.

Iyer et al. [11] replicated the work of Tsay et al. [14] and extended it by adding and evaluating factors that quantified certain personality traits. First, they implemented the model by Tsay et al. and the previous findings were largely replicated. Then, personality factors measuring the openness, conscientiousness, extraversion, agreeableness, and neuroticism of the requester and closer were added to the model. For requesters, higher openness and conscientiousness increased acceptance, while higher extraversion correlated with more rejection. For closers, higher conscientiousness, neuroticism, and extraversion increased acceptance, respectively. Finally, they modeled the personality differences between the requester and the closer. The difference in conscientiousness, neuroticism, and extraversion had the strongest positive association with acceptance, respectively.

Ortu et al. [18] studied the influence of affect and politeness on acceptance. This was achieved by processing issue and pull request comments and modeling the affect and politeness factors with four confounding factors. Higher levels of arousal, dominance, and anger received by the submitter were associated with rejection, while higher levels of joy and valence were associated with acceptance. Politeness was insignificant. They also measured the predictive power of each factor. The most important factors for acceptance were valence and whether the submitter is a developer, and the most important factors for rejection were dominance, arousal, and anger. Lastly, two classifiers were implemented with varying sets of the affect factors to evaluate if they can improve acceptance predictions. Affect factors were found to improve the prediction significantly in several measures.

A few studies have focused specifically on rejected PRs. Soares et al. [19] studied why PRs by core team members were rejected in a high PR acceptance context. Association rules were used to extract several factors, including commonly used technical factors and other project-specific ones, and identify patterns. Submitter experience, locality, and PR size strongly influenced rejection in general, some factors increased rejection even more in conjunction with each other, and project-specific factors like test inclusion were significant. Gousios et al. [4] reported the most common reasons for rejection by manually

inspecting a PR sample. Respectively, these were concurrent modifications in the project, e.g., superseded PRs, uninteresting changes, e.g., superfluous PRs, incorrect implementations, and failing process and quality requirements, e.g., PRs lacking tests. Steinmacher et al. [3] found similar reasons using the same method. Silva et al. [13] studied how technical debt affects rejection by categorizing PRs according to their discussion. The most common reasons respectively were design debt, test debt, and project convention debt, similar findings to the two aforementioned works.

2.2.2 Understanding the pull request lifetime

Compared to the acceptance of PRs on GitHub, relatively fewer studies have contributed to understanding the lifetime. Again, the most prevalent method is modeling, which has been based on a multitude of factors, such as technical [6], social [5], CI [20], and code quality [7] factors. This section summarizes well-known studies, studies that add to previous work, and studies with novel contributions.

Beyond PR acceptance, Gousios et al. [4] also contributed to the matter of the lifetime. Their execution was similar to what has already been explained in Section 2.2.1, except the models predicted the time class of a PR: merged within an hour, a day, or beyond. Again, the Random Forest algorithm was most successful in the predictions and thus used to find the dominant factors. The three strongest factors were the submitter's previous acceptance rate, project size, and project test coverage. No factors were dominant, unlike when predicting the likelihood of PR merges. Bernardo et al. [21] also found submitter experience to decrease review time.

Kononenko et al. [6] sought to find the factors that affected the PR review time PR in a case study of a commercial project. Using multiple regression modeling, the submitter's affiliation was found to have the strongest effect on the delay, where developers of the core team and another actively contributing team received the quickest reviews. Lengthy discussion, both in the PR thread and on its source code, led to longer delays, which was the second strongest effect. Small PRs authored by experienced contributors had shorter delays. Their findings were slightly different from Gousios et al. [4], suggesting that the importance of some factors depends on the project.

Multiple studies have considered CI in relation to the PR lifetime. In a follow-up of a previous study [22], Yu et al. [5] studied what factors influenced the closing time of PRs in 400 GitHub projects that used a CI service. When modeling without CI factors, they arrived at similar findings to Kononenko et

al. [6]. When considering CI, CI was found to have a large effect, yet smaller than the usual dominant factors. Bernardo et al. [21] examined the impact of adopting CI on the delivery time of PRs. They achieved this by measuring the importance of 13 factors before and after a project adopted CI. The same three factors were the most influential before and after CI adoption. These were the number of open PRs at submission, the moment when the PR is merged compared to other merged PRs in the release cycle, and the submitter's experience. Guo and Leitner [20] replicated this study exactly and conceptually and introduced one new factor. Both approaches achieved similar results with minor differences, concluding that the results were generalizable. The new factor, i.e., the time when a PR is submitted in the release cycle, had the most explanatory power after CI adoption.

de Lima Júnior et al. [23] beat the PR lifetime prediction model by Yu et al. [22] by experimenting with different regression techniques. The M5P and SMOReg algorithms performed the best out of their algorithm selection. However, the authors pointed out the limited usefulness of their approach due to the high error rate. To address this, they executed another experiment with classification algorithms to predict a PR's time class, as opposed to its precise lifetime, similarly to Gousios et al. [4]. Random Forest performed the best in this experiment. The study also put forth 25 new factors, of which 14 were original and 11 had been used to predict things other than lifetimes. Nine of the new factors had significant predictive power.

As elaborated on in Section 2.2.1, Zou et al. [7] performed an exploratory study on Java projects to see how code style affects PRs. Code style inconsistency was measured between the files of a PR and the original files through several metrics. These were modeled along with other confounding factors. Overall, style inconsistency had a small positive effect on closing time, meaning that a large inconsistency correlated with a long lifetime. Specifically, inconsistency w.r.t. maximum line lengths and writing of method or constructor Javadocs affected closing time the most.

Moreira et al. [24] took a novel approach by extracting association rules to identify and justify behavior in terms of the PR lifetime. They studied a wide range of factors based on their prior works, one of which [19] is summarized in Section 2.2.1, and also analyzed qualitative aspects to gain a deeper knowledge of some patterns. It was shown that smaller PRs had a shorter life span, corroborating with Kononenko et al. [6], and that shorter life spans increased acceptance. Another influential factor was whether the PR touches specific files and directories, e.g., recently modified and critical areas. Furthermore, the submitter's experience, status, and social relations all impacted the life-

time. Interestingly, there was evidence that PRs filed by core team members tended to last longer, at odds with previous findings [4]. Finally, they found lengthy discussions and certain reviewers to increase lifetime.

2.2.3 Understanding pull request recommendation

Multiple strategies have been devised that recommend PRs for reviewers on GitHub to increase productivity. The strategies include recommendation based on response [8], acceptance [25], quick reviewing [1] and lifetime [9]. Some models have been packaged as tools and deployed to reviewers [8, 26, 9]. This section summarizes well-known studies, studies that add to previous work, and studies with novel contributions.

One of the first PR recommendation tools was prototyped by van der Veen et al. [8]. The tool, called PRioritizer, informed reviewers about the PRs most likely to receive a user update the following day. When an event arrives to the watched repository, the open PRs are fetched, stored in a local git clone, and processed. The results are fed to a Random Forest model which creates an ordering of the PRs and a visualizer uses the output to present recommendations with functionality such as filtering and sorting. In a preliminary user study, 450 GitHub projects were invited to use PRioritizer and share their experiences through a survey. While the developers agreed that the service would not cause overhead and provided an easy-to-comprehend overview of the PR status, the reception of the recommendation was mixed and they requested more insight into how the ranking worked.

Noting that the PR lifetime had not been explored for recommendations, Zhao et al. [1] proposed a learning-to-rank (LtR) approach to recommending PRs that can be quickly reviewed. PRs were categorized according to their outcome and time to review and merge. Then, six LtR algorithms were trained to predict the category of the PRs at different time intervals. Random Forest outperformed the other algorithms as well as two baseline criteria, i.e., first-in-first-out and smallest-size-first, that reviewers had previously shared [2]. 18 factors were used and their respective importance was quantified by measuring the decrease in prediction accuracy when excluding each factor from training. Social factors were found to be most important, followed by the number of commits and code complexity. Reviewers were surveyed to evaluate the usefulness of their approach, of which the majority agreed the ranking could increase efficiency in terms of reviewing PRs.

As PRioritizer [8] did not take acceptance into account, Azeem et al. [25] developed AR-Prioritizer (Acceptance and Response based Prioritizer). Rec-

ommendations were based on the likelihood of a PR being updated the following day, like with PRioritizer, but the likelihood of acceptance was added as well. Specifically, AR-Prioritizer recommended PRs based on the descending average probability of response and acceptance. The probabilities were based on two models which considered 56 factors pertaining to the project, PR, submitter, and reviewer dimensions. Both models used XGBoost which was found to outperform three other models including Random Forest. Two baseline models were implemented to benchmark the efficacy of AR-Prioritizer: PRioritizer for comparing response predictions, and one model by Gousios et al. [4] for comparing acceptance predictions. Their tool outperformed both, concluding that their approach was promising.

Azeem et al. [27] devised an approach to recommend actions to take on PRs, called CARTESIAN. CARTESIAN recommended one of three PR actions: accept the PR without discussion, respond to it because it is likely to become accepted after discussion, or reject it. While the approach was novel, the models and factors used are the same as for AR-PRioritizer [25]. To find the most important factors, the classifier was run multiple times and the relative contribution was recorded for each factor. 31 out of 56 factors were found to be necessary to maintain the prediction capability. The most important factors were the number of review and discussion comments, whether the submitter was a contributor, and the number of participants in the discussion, respectively. The usefulness of CARTESIAN was assessed by ranking the PRs within their predicted class, i.e., accept, respond, or reject, and comparing the rankings with the same baselines used by Zhao et al. [1]. CARTESIAN outperformed both baselines. After a manual inspection of the ranked PRs, it was revealed that bug fixes and new features are given high priority, corroborating with previous findings [2].

Maddila et al. [26] developed, deployed, and evaluated the PRLifetime Service, a tool to accelerate overdue PRs to completion or abandonment. It used the Azure DevOps API, the source control system used by Microsoft developers, and a Gradient Boosting model to infer PR lifetimes based on 28 factors. When a PR exceeds its predicted lifetime, the service notified the appropriate users in an attempt to expedite the final merge decision. When deployed to reviewers, the average PR completion time was reduced by 43%, indicating success. However, users complained about unneeded notifications, e.g., they would be notified while waiting for the other party to respond to a PR. With this shortcoming in mind, Maddila et al. [9] created another tool called Nudge with the same goal and similar execution with various quality-of-life changes. First, they took measures to detect and not send notifications on active PRs.

Second, if a notification was to be sent, they determined and notified specifically the user who blocked progress. They also added the capability to collect feedback through its interface. When deployed and evaluated, Nudge reduced the average PR completion time by 61%, an improvement over the PRLifetime Service, and 73% of the notifications were positively acknowledged by users.

2.2.4 Understanding pull request user actions and opinions

Some works have focused on the actions and opinions of PR users on GitHub with regard to development and review. This section summarizes such works, ranging from large-scale surveys to case studies.

Gousios et al. [10] sought to answer how submitters prepare PRs with an online survey on top submitters to active open-source software projects on GitHub. Respondents were asked what they do before and after developing their PRs and how they evaluate them. They received 645 responses, the majority of which identified as working for the industry with more than seven years of experience in software development. Before coding, most submitters tried to increase project awareness, i.e., understand what is being worked on and what needs to be done, except for the most experienced users who are already familiar with the project status. After coding, most submitters did not check for overlapping work that may have occurred during coding. Instead, they formatted their PR according to guidelines and ran tests against it. The most common methods of self-evaluation were testing, code review, and static analysis. In self-code review, they evaluated compliance with both PR and coding guidelines, clarity, atomicity, and mergeability.

Similar to the aforementioned study, Gousios et al. [2] conducted another survey on active GitHub users, this time focusing on integrators, i.e., users who make merge decisions. Top integrators were asked how they review PRs, decide to merge them, and evaluate their quality. They received 749 responses. Most of the participants were project owners working for the industry with more than seven years of experience in software development. During code review, 75% of integrators used inline code comments, 8% used commit comments, and only 1% explicitly mentioned using a different tool when reviewing. 42% of integrators delegated reviews to other users if they are not familiar with the code and at least 3% reported that reviews from multiple integrators are required to ensure quality. The most important factors for acceptance were quality, project fitness, and test inclusion. Integrators judged quality mainly

by inspecting conformance to project and coding standards, test coverage and results, and documentation.

Steinmacher et al. [3] uniquely focused on reasons for PR rejection in two surveys. Their first survey was on *quasi-contributors*, i.e., users who had only faced rejection in the specific project, which they found made up 70% of total submitters. Respectively, the top five self-perceived reasons for rejection were superseded or duplicated PRs, different visions or opinions, unneeded or irrelevant PRs, lack of interest from integrators, and suboptimal implementations. They received 335 answers and their second survey which was for integrators received 21 answers. The reasons for rejection were different from the integrator's perspective. The most common were unneeded or irrelevant PRs and failing to follow guidelines. Interestingly, nobody mentioned a lack of interest from integrators. Lastly, two respondents perceived small fixes, e.g., fixing typos, as noisy and thus undesirable.

Two studies executed smaller surveys to complement their other results. First, 16 developers shared how they judge and review PRs in a case study by Kononenko et al. [6]. The most influential aspects on the merge decision were the number of participants in the discussion, the number of modified files, and the number of lines changed. Respondents typically conducted code reviews by checking if the tests succeed, evaluating the quality of the PR, and trying to understand its scope. They judged PR quality mainly according to its description, code quality, and complexity. Finally, they judged PR review quality according to the reviewer's experience and response time, and lines of code of the PR. Zampetti et al. [17] conducted another minor survey where 13 practitioners of CI shared its relation to PRs. The respondents agreed that the first and last PR build status both influenced acceptance. When a PR build fails, the most common CI-related action was discussing test failure, which is then isolated or addressed through private builds or re-executing the CI build. Other actions were far less common, e.g., pointing out code style errors when the static analysis check fails.

2.2.5 Pull request factors modeled

Table 2.1 lists factors that have been previously modeled to predict the acceptance or lifetime of PRs or to recommend PRs. The typeface of a particular citation marks the statistical significance of the factor in the experiments performed in cited studies. Read the caption of the table for specifics. Many factors, descriptions, and categories have been renamed and merged for clar-

ity. Also note that the factors may have been calculated at different times, typically at the time of PR creation, closure, or time of data collection.

Table 2.1 – Factors used by related works to model pull requests. Citations in bold means the factor was statistically significant in at least one experiment ($p \leq 0.05$). Citations in regular font means the significance of the factor was not measured. Citations in italics means the factor was statistically insignificant in all experiments ($p > 0.05$). Similar factors have been renamed or merged.

Factor	Description	Studies on acceptance	Studies on lifetime	Studies on recommendation
Factors pertaining to PR changes				
Changed_Files	# of files touched by the PR	**[14]** **[11]** **[16]** [17] [7] [4] [15]	[7] [4] [23] [21] [20]	[26] [9] [25] [27] [8] [1]
PR_Commits	# of commits in the PR	**[5]** **[18]** **[16]** [17] [4] [15]	**[5]** **[22]** [4] [23] [21] [20] *[6]*	[25] [27] [8] [1]
Lines_Changed	# of lines changed by the PR	**[11]** **[14]** **[6]** [7] [4]	**[6]** [7] [4] [23] [21] [20]	[26] [9] [25] [27] [1]
Added_Lines	# of lines added by the PR	**[5]** **[16]** [17] [15]	**[5]** **[22]**	[8]
Deleted_Lines	# of lines deleted by the PR	**[5]** [17] [15] *[16]*	**[5]** **[22]**	[8]
Changes_Config_Or_ Settings	Is the PR modifying any configuration files or settings?			[26] [9]
Classes_Changed	# of classes changed			[26] [9]
Conditionals_Touched	# of conditional statements changed			[26] [9]
File_Types	# of file types changed			[26] [9]
Is_CSProj_Modified	Is the .csproj file being touched?			[26] [9]
Lines_Changed_Per_Class	# of lines changed per class			[26] [9]
Loops_Touched	# of loops changed			[26] [9]
Methods_Changed	# of methods changed			[26] [9]
Paths_Changed	# of file paths touched			[26] [9]
References_Or_ Dependencies_Changed	# of references/dependencies on other libraries/projects changed			[26] [9]
Added_Files	# of files added in the PR		[23]	
Changed_Doc_Files	# of doc files touched by the PR		[23]	
Changed_Other_Files	# of other files touched by the PR		[23]	
Changed_Source_Files	# of source files touched by the PR		[23]	
Deleted_Files	# of files deleted by the PR		[23]	
Total_Churn	Total churn in the PR			[9]
Factors pertaining to PR metadata				
Contains_Fix	Does the PR refer to a bug, issue, or other PR?	**[5]** **[16]** [4] [17]	**[5]** **[22]** [4] [23]	[26] [9] [25] [27] [8] [1]
Hotness	How much a PR changes recently modified code (measured in various ways)	**[5]** [4]	**[5]** **[22]** [4] [23]	[1]
Description_Length	Word count of the description		[21] [20]	[26] [9] [25] [27] [1]

Continued on next page

Table 2.1 continued from previous page

Factor	Description	Studies on acceptance	Studies on lifetime	Studies on recommendation
Has_Mention_Tag	Does the PR title or description contain an @-mention tag?	*[5]*	**[5]** **[22]** [23]	[25] [27]
PR_Age	Age of the PR	**[16]** [17] [15]		[25] [27] [8]
Title_Length	Word count of the title			[26] [9] [25] [27] [1]
Title_And_Description_Length	Word count of the PR title and description	**[5]**	**[5]** **[22]** [23]	
Submitted_On_Friday	Was the PR submitted on Friday?	*[5]*	**[5]** **[22]** [23]	
Day	The day of PR submission			[26] [9] [25] [27]
Is_Intra_Branch	Are the source and target repos the same?	[17]		[25] [27] [8]
Adds_Feature	Does the PR add a feature?			[26] [9]
Assignees	# of users assigned to the PR			[25] [27]
Business_Hours	Was the PR submitted during business hours?			[26] [9]
Contains_Conflict	Does the word “conflict” appear in the PR comments?	[4]	[4]	
Description_Embedding	Embedding of the description			[25] [27]
Events	# of events on the PR		[21] [20]	
Has_Merge_Changes	Is the PR making merge changes?			[26] [9]
Has_Stack_Trace	Is there a stack trace in the PR description?		[21] [20]	
Is_Deprecating	Is the PR deprecating code?			[26] [9]
Is_Mergeable	Is the PR in a mergeable state?			[25] [27]
Is_Refactor	Is the PR refactoring code?			[26] [9]
Labels	# of labels in the PR			[25] [27]
Last_Commit_Mentions	Does the last commit mention a user?	[17]		[8]
Points_To_Issue_Or_PR	Does the PR point to an issue or another PR?			[25] [27]
Queue_Rank	When the PR is merged in the release cycle		[21] [20]	
Title_Embedding	Embedding of the title			[25] [27]
Dependency	Does any repo the submitter has contributed to depend on the repo the PR was submitted in?	**[16]**		
Readability	Readability of the PR title, description, and commit messages			[1]
Release_Cycle_Timing	When the PR is submitted in the release cycle		[20]	
Factors pertaining to the repository				
Repo_Open_PRs	# of open PRs in the repo	**[5]** [7]	**[5]** **[22]** [7] [23] [21] [20]	[26] [9] [25] [27]
Team_Size	# of total or active core members in the repo	**[14]** [4] [7] *[5]* *[11]*	**[5]** **[22]** [4] [23]	[25] [27]
Repo_Age	Age of the repo	**[14]** **[5]** **[11]** [7]	**[5]** **[22]** [7] [23]	[25] [27]
Stars	# of stars of the repo	**[14]** **[11]**	[23]	[25] [27]
Repo_Merge_Rate	% of merged PRs in the repo	**[16]** [15]		[25] [27]
Forks	# of forks of the repo	[7]	[7]	[25] [27]
Repo_PRs	# of PRs submitted to the repo	**[16]** [15]	[23]	

Continued on next page

Table 2.1 continued from previous page

Factor	Description	Studies on acceptance	Studies on lifetime	Studies on recommendation
Repo_Size	Lines of code in the repo	[4]	[4] [23]	
Contributors	# of contributors in the repo			[25] [27]
Domain	Domain of the repo			[25] [27]
External_Commits_Ratio	% of recent commits in this repo coming from external users	[4]	[4]	
Lifetime_For_Same_Paths	Average lifetime of PRs that touch the same file paths as this PR			[26] [9]
Open_Issues	# of open issues in the repo			[25] [27]
Repo_Comments_Per_Merge	Average # of comments on PRs merged into the repo			[25] [27]
Repo_Comments_Per_PR	Average # of comments on the PRs of the repo			[25] [27]
Repo_Files_Changed_Per_PR	Average # of files touched by every PR in the repo			[25] [27]
Repo_Language	Programming language of the repo			[25] [27]
Repo_Lines_Added_Rate	# of line additions per week in the repo			[25] [27]
Repo_Lines_Changed_Per_PR	Average # of lines changed in every PR in the repo			[25] [27]
Repo_Lines_Changed_Weekday_X	Lines changed per day on each weekday (one factor per weekday)			[25] [27]
Repo_Lines_Deleted_Rate	# of line deletions per week in the repo			[25] [27]
Repo_Merge_Delay	Average time to merge a PR in the repo			[25] [27]
Repo_PR_Commits	Average # of commits per PR in the repo			[25] [27]
Repo_PR_Lifetime	Average lifetime of a PR in the repo			[25] [27]
Repo_Test_Lines_Per_KLOC	# of test lines per one thousand lines of code in the repo	[4]	[4]	
Watchers	# of watchers of the repo			[25] [27]
PR_Commits_Ratio	% of recent commits in the repo made by PRs		[23]	
Factors pertaining to the submitter				
Submitter_PRs	# of PRs by the submitter	**[6]** [4]	**[6]** [4] [23] [21] [20]	[25] [27]
Submitter_Merge_Rate	% of merged PRs by the submitter	[17] [4]	**[22]** [4] [23]	[25] [27] [8] [1]
Submitter_Merged_PRs	# of merged PRs by the submitter	[7]	[7]	[25] [27]
Submitter_Merged_NPM_PRs	% of merged PRs across NPM projects by the submitter	**[16]** [15]		
Submitter_NPM_PRs	# of PRs across NPM projects by the submitter	**[16]** [15]		
Submitter_Project_Contributions	# of projects the submitter has contributed to	**[16]** [15]		
Submitter_Total_Commits	# of commits across all projects by the submitter	**[16]** [15]		
Submitter_Average_PR_Lifetime	Average lifetime of PRs by the submitter			[26] [9]
Submitter_Average_PR_Release_Time	Average lifetime of released PRs by the submitter		[21] [20]	
Submitter_Closed_PRs	# of closed PRs by the submitter			[25] [27]
Submitter_Closed_Rate	% of closed PRs by the submitter			[25] [27]
Submitter_Commit_Rate	% of submitters' commits before the PR	[17]		[8]

Continued on next page

Table 2.1 continued from previous page

Factor	Description	Studies on acceptance	Studies on lifetime	Studies on recommendation
Submitter_Repos	# of public repos by the submitter			[25] [27]
Submitter_Time_At_Microsoft	Time spent by the submitter at Microsoft			[26] [9]
Submitter_Time_In_Team	Time spent by the submitter in the current team			[26] [9]
Submitter_Time_Since_First_Activity	Time since the submitter's first activity in the repo			[26] [9]
Submitter_Has_Merged_PR	Does the submitter have a merged PR in the repo?	**[16]**		
Submitter_Prior_Comments	# of comments by the submitter	**[18]**		
Submitter_Age	Age of the submitter's account		[23]	
Submitter_Blobs	# of blobs created by the submitter	[15]		
Submitter_Commits	# of commits in this project by the submitter		[23]	
Submitter_Commit_Comments	# of comments on commits in this project by the submitter		[23]	
Factors pertaining to code review				
Discussion	# of discussion comments on the PR	**[14]** **[5]** **[11]** **[16]** [17] [15]	**[5]** **[22]** [23] [21] [20]	[8]
First_Human_Response	Time until the PR receives a reviewer response	**[5]**	**[5]** **[22]** [23]	[25] [27]
Review_Comments	# of comments attached to code on the PR	**[16]** [17] [15]		[25] [27] [8]
Participants	# of participants in the PR discussion	[4]	[4]	[26] [9] [25] [27]
Discussion_And_Review_Comments	# of discussion and review comments on the PR	[4]	[4]	[25] [27]
Integrator_Availability	Average time until one of the top two integrators is available	**[5]**	**[5]** **[22]**	
Code_Commenting_Reviewers	# of reviewers who commented on source code	**[6]**	**[6]**	
Commenting_Reviewers	# of reviewers participating in the discussion	**[6]**	**[6]**	
Average_Time_Between_Comments	Average time between comments on the PR		[21] [20]	
Discussion_And_Review_Words	Word count of the PR's discussion and review comments			[25] [27]
Lifetime_After_First_Response	Time between the first reviewer response and PR closure			[25] [27]
Response_Received	Did a reviewer respond to the PR?			[25] [27]
Wait_Time	Waiting time of the PR			[25] [27]
Submitter_Discussion	# of comments by the submitter	*[6]*		
Factors pertaining to social aspects				
Submitter_Status	Is the submitter a contributor, core member, or user with commit access to the repo?	**[14]** **[5]** **[11]** **[18]** [17]	**[5]** **[22]** [23]	[25] [27] [8] [1]
Submitter_Followers	# of followers the submitter has	**[14]** **[5]** *[11]*	**[5]** **[22]** [23]	[25] [27] [1]
Prior_Interaction	% of team members that interacted with the submitter recently	**[5]**	**[5]** **[22]**	
Submitter_Events	# of events the submitter has participated in for this project	**[14]** **[11]**		[1]

Continued on next page

Table 2.1 continued from previous page

Factor	Description	Studies on acceptance	Studies on lifetime	Studies on recommendation
Submitter_Issues	# of issues reported by the submitter	**[18]** [7]	[7]	
Submitter_Follows_Closer	Does the submitter follow the closing reviewer?	**[14]** **[11]**		
Submitter_Affiliation	The organization the submitter is affiliated with	**[6]**	**[6]**	
Submitter_Contributions	# of contributions by the submitter			[25] [27]
Submitter_Follows	# of users the submitter follows			[25] [27]
Submitter_Follows_Reviewer	Does the submitter follow a reviewer?		[23]	[1]
Reviewer_Follows_Submitter	Does a reviewer follow the submitter?		[23]	
Submitter_Follows_Project	Does the submitter follow the project?		[23]	
Submitter_Issue_Comments	# of comments in issues by the submitter		[23]	
Submitter_Issue_Events	# of events in issues the submitter has participated in		[23]	
Submitter_PR_Comments	# of comments in PRs by the submitter		[23]	
Submitter_PR_Events	# of events in PRs the submitter has participated in		[23]	
Factors pertaining to Continuous Integration				
CI_Latency	Time between PR submission and the last commit tested by CI	**[5]**	**[5]** **[22]**	
Has_CI_Failure	Did any of the CI builds fail on this PR?	**[5]**	**[5]** *[22]*	
Build_Fail_Rate	% of failed builds across the PR	[17]		
Builds	# of builds across the PR	[17]		
CI_Used	Does the PR use CI?		[23]	
First_Build_Status	First PR build status	[17]		
Last_Build_Status	Last PR build status	[17]		
Factors pertaining to personality and emotion				
Submitter_Agreeableness	Agreeableness of the submitter	*[11]*		
Submitter_Conscientiousness	Conscientiousness of the submitter	**[11]**		
Submitter_Extraversion	Extraversion of the submitter	**[11]**		
Submitter_Neuroticism	Neuroticism of the submitter	*[11]*		
Submitter_Openness	Openness of the submitter	**[11]**		
Closer_Agreeableness	Agreeableness of the closer	*[11]*		
Closer_Conscientiousness	Conscientiousness of the closer	**[11]**		
Closer_Extraversion	Extraversion of the closer	**[11]**		
Closer_Neuroticism	Neuroticism of the closer	**[11]**		
Closer_Openness	Openness of the closer	**[11]**		
Agreeableness_Diff	Difference in agreeableness between the submitter and closer	**[11]**		
Conscientiousness_Diff	Difference in conscientiousness between the submitter and closer	**[11]**		
Extraversion_Diff	Difference in extraversion between the submitter and closer	**[11]**		
Neuroticism_Diff	Difference in neuroticism between the submitter and closer	**[11]**		
Openness_Diff	Difference in openness between the submitter and closer	*[11]*		

Continued on next page

Table 2.1 continued from previous page

Factor	Description	Studies on acceptance	Studies on lifetime	Studies on recommendation
Submitter_Anger	Anger expressed by the submitter in issues they did not create	*[18]*		
Submitter_Arousal	Arousal expressed by the submitter in issues they did not create	**[18]**		
Submitter_Dominance	Dominance expressed by the submitter in issues they did not create	**[18]**		
Submitter_Joy	Joy expressed by the submitter in issues they did not create	**[18]**		
Submitter_Love	Love expressed by the submitter in issues they did not create	*[18]*		
Submitter_Politeness	Politeness expressed by the submitter in issues they did not create	*[18]*		
Submitter_Sadness	Sadness expressed by the submitter in issues they did not create	**[18]**		
Submitter_Valence	Valence expressed by the submitter in issues they did not create	**[18]**		
Submitter_Received_Anger	Anger received by the submitter on their own issues	**[18]**		
Submitter_Received_Arousal	Arousal received by the submitter on their own issues	**[18]**		
Submitter_Received_Dominance	Dominance received by the submitter on their own issues	**[18]**		
Submitter_Received_Joy	Joy received by the submitter on their own issues	**[18]**		
Submitter_Received_Love	Love received by the submitter on their own issues	*[18]*		
Submitter_Received_Politeness	Politeness received by the submitter on their own issues	*[18]*		
Submitter_Received_Sadness	Sadness received by the submitter on their own issues	*[18]*		
Submitter_Received_Valence	Valence received by the submitter on their own issues	**[18]**		
	Factors pertaining to code quality			
Has_Test_Files	Does the PR have test files?	**[5]** **[11]**	**[5]** **[22]** [23]	[1]
Has_Test_Code	Does the PR have test code?	**[14]** [17] *[16]*		[8]
Test_Churn	# of test lines changed in the PR	[4]	[4] [23]	[1]
Style_Diff*	Code style difference between the PR files and original files	[7]	[7]	
Code_Comments_Added	# of source code comments added by the the PR			[1]
Cyclomatic_Complexity_Added	Cyclomatic complexity of the code added in the PR		[1]	
Cyclomatic_Complexity_Deleted	Cyclomatic complexity of the code deleted in the PR		[1]	
Has_PMD_Issue_X	Does PMD detect code quality issue X in this PR? (One factor per issue)	[12]		

* Zou et al. [7] also modeled the 37 individual kinds of style differences this factor was based on. Six of them were insignificant.

Chapter 3

Study design

In this chapter, we explain the design of our mixed-method study. First, we motivate our study (Section 3.1). Then, we present our quantitative study which uses machine learning on GitHub data to answer RQ1 and RQ2 (Section 3.2), and present our qualitative study which uses a survey for GitHub users to answer RQ3 and RQ4 (Section 3.3). The scripts and data produced in this work are publicly available for future research (Section 1.4).

3.1 Motivation

As evidenced in Section 2.2, many studies have modeled pull request (PR) acceptance and conducted surveys on GitHub users. However, the models have not been evaluated in terms of performance change when tested on open PRs, which is the real use case scenario of tools that recommend PRs for reviewers. Moreover, the surveys have not completely captured all aspects of PR usage. To showcase the novelty of our work, we summarize key aspects of this work in contrast to the related ones.

Table 3.1 contrasts the quantitative part of this work. The lowest common denominator between this and the related works is the modeling of PR acceptance as well as measuring the importance of the factors. Our work diverges by comparing the prediction of open PRs to closed PRs. This is valuable to developers of PR recommendation tools because they typically work with closed PRs, since the merge status needs to be known for model training, but models must make predictions on open PRs to be useful. Dey and Mockus [15] observed this previously and thus sought to answer how their models fared against open PRs, but we argue that their approach was flawed. Instead of predicting on open PRs, closed PRs were used with *some* time-sensitive factors

Table 3.1 – Comparison between this work and related works that modeled pull request acceptance. Items marked with asterisks are elaborated on in Section 3.1.

Work	Considered open PRs	Number of PRs	Number of factors	Number of algorithms
Gousios et al. [4] (2014)	×	170K	15	6
Tsay et al. [14] (2014)	×	660K	11	1
Yu et al. [5] (2016)	×	100K	20	1
Kononenko et al. [6] (2018)	×	1K	7	1
Iyer et al. [11] (2019)	×	500K	26	1
Zampetti et al. [17] (2019)	×	190K	18	1
Zou et al. [7] (2019)	×	50K	*45	1
Dey and Mockus [15] (2020)	*×	480K	14	5
Dey and Mockus [16] (2020)	×	470K	17	2
Ortu et al. [18] (2020)	×	160K	20	2
Lenarduzzi et al. [12] (2021)	×	40K	*253	8
This work (2022)	✓	*100K	19	7

removed. However, some temporal factors remained, e.g., the number of commits, hence the need for further investigation. Our approach can truly recreate open PRs based on their closed counterparts to enable the analysis of PRs in various states. We focused on three specific states, which also means that the size of our data set is arguably tripled, making it more comparable to the related works. The number of factors we studied is also comparable with the exceptions of Zou et al. [7] and Lenarduzzi et al. [12] which both modeled individual code quality flaws as factors.

Table 3.2 contrasts the qualitative part of this work. While this and the related works all surveyed GitHub users, we distinguish ourselves by investigating the submitter response and reviewer opinion specifically in the first and final review. Our survey was also uniquely made for both submitters and reviewers to cover the interplay between both parties. Note that this means that our data set size nearly doubles, since the vast majority of our respondents answered both the submitter and reviewer sections. Not in this table but worthy of mention is also the difference w.r.t. the number of multiple-choice options in each question, where our work is most thorough to allow for a more nuanced study.

Table 3.2 – Comparison between this work and related works that surveyed pull request users.

Work	Inquiries of survey(s) w.r.t. PR usage	Invited demographic	Number of responses
Gousios et al. [2] (2015)	• Reviewer response to PRs • Reviewer opinion on PR quality	Experienced reviewers	749
Gousios et al. [10] (2016)	• Submitter preparation of PRs • Submitter opinion on PR quality	Experienced submitters	760
Kononenko et al. [6] (2018)	• Reviewer response to PRs • Reviewer opinion on PR quality • Reviewer opinion on review quality	Developers on the same project	16
Steinmacher et al. [3] (2018)	• Submitter opinion on PR rejection • Reviewer opinion on PR rejection	Unsuccessful submitters and any reviewer	356
Zampetti et al. [17] (2019)	• Opinion on CI on PR quality • Actions related to CI on PRs	CI practitioners	13
This work (2022)	• Submitter preparation of PRs • Submitter response to feedback • Reviewer response to PRs • Reviewer opinion on PR quality in the first and final review	Any submitter and reviewer	204

3.2 Quantitatively studying pull request acceptance (RQ1 and RQ2)

In this section, we describe the experiments carried out using GitHub data and machine learning to answer RQ1 and RQ2. In Section 3.2.1, we define the PR timeline, the fundamental piece to allow the time-sensitive experiments. Section 3.2.2 establishes how to generate timelines with GitHub's API. We devise and apply criteria to create the set of factors for investigation in this work in Section 3.2.3. In Section 3.2.4, we detail the process of selecting and gathering PR data for the experiments. Finally, we describe the machine learning experiments executed to answer RQ1 and RQ2 in Section 3.2.5.

3.2.1 Defining the pull request timeline

RQ1 and RQ2 can only be answered by analyzing PRs in a way that respects their changes over time. For example, the number of commits, labels, and comments almost certainly change. By storing the events that occurred on a

PR in a *timeline*, we can recreate the state of a PR at any point in time by stepping through the events and updating the representation of the PR accordingly. Temporal experiments can therefore be executed by carefully choosing and examining multiple states of PRs. We chose to consider three states: *ready* (ready for review), *middle* (halfway complete), and *closure* (merged or non-merged). Figure 3.1 showcases a timeline and underlines these states. The following subsections provide a rigorous definition of each state.

Figure 3.1 – Timeline of a pull request. The events are named according to GitHub's convention. The ready, middle, and closure states have been marked.

	Time	Event	Actor
	10:00	renamed	Submitter
	10:00	labeled	Bot
	10:02	cross-referenced	Submitter
	10:04	review_requested	Submitter
Ready	11:00	reviewed	Reviewer A
	11:01	mentioned	Reviewer A
	11:30	commented	Reviewer B
	14:00	committed	Submitter
Middle	14:02	review_requested	Submitter
	16:00	reviewed	Reviewer A
	16:02	merged	Reviewer A
	16:02	closed	Reviewer A
Closure			

The ready state

For our first state of choice, it made sense to capture what the PR looked like early in its lifetime. Because submitters may perform actions after opening their PR and before another person begins review, e.g., making small adjustments and requesting a review[1], we chose to focus on when the PR is ready for its first review, rather than when it was submitted, to more accurately capture the intention of the PR. GitHub has a useful feature for marking PRs as

[1]Example: `https://github.com/envoyproxy/envoy/pull/9480`. Last access 2022-11-30.

ready for review[2], however, it is seldom used[3]. Instead, we define *ready* as the PR state when a person other than the submitter first shows activity on the timeline. See Figure 3.1 for an example of the ready state. The aforementioned definition is a simplification to improve understanding and we provide the rigorous definition below.

> Definition: *ready* is the pull request state right before another *participant* appears on the timeline. If the submitter is the only participant, the ready state is located precisely one-third through the timeline, truncating the index if necessary.

In the rigorous definition above, we define a *participant* as a human actor on the PR. This minimizes two problems that would otherwise prematurely mark the ready state. First, we only count humans as participants because bots may perform actions on a PR before the submitter has had any chance to fully prepare it. Bots are identified through a flag in GitHub's data, however, it is not perfectly reliable[4] and therefore the username is matched with "bot" to catch additional cases. Second, we discount some events[5] when finding participants because they can be misleading. For instance, @-mentioning a user creates an invisible event for which the actor is the mentioned user, rather than the user who truly caused the event.

The closure and middle states

The last two states are easier to define. *Closure* is the PR state the moment it is closed and thus merged or non-merged. For PRs that have been reopened and closed again, closure is on the final close. *Middle* is the PR state halfway between ready and closure, counted by the number of events rather than timestamps. If the halfway point is a non-integer, we truncate it. The middle is located through the number of events because PRs tend to change sporadically as opposed to continuously over time, usually because one party submits a cluster of events and waits for the other to respond[6]. For example, in the timeline

[2]Docs: `https://docs.github.com/en/developers/webhooks-and-events/events/issue-event-types#ready_for_review`. Last access 2022-11-30.
[3]Used by 2% of PRs in a sample of 1,000.
[4]Example: `https://api.github.com/users/squash-labs`. Last access 2022-11-30.
[5]The events are {automatic_base_change_failed, automatic_base_change_succeeded, mentioned, subscribed}. Docs: `https://docs.github.com/en/developers/webhooks-and-events/events/issue-event-types`. Last access 2022-11-30.
[6]Example: `https://github.com/obsproject/obs-studio/pull/2171`. Last access 2022-11-30.

presented in Figure 3.1, the ready state is located after event 4, the closure state captures all 12 events, and the middle state is halfway, i.e., $\lfloor (4 + 12)/2 \rfloor = 8$.

3.2.2 Creating the pull request timeline

We have argued that a timeline is necessary for our experiments and the next challenge was to devise a way of generating the timeline of any PR. Conveniently, GitHub's Timeline events API[7] offered a recall of any PR, including events such as committing, labeling, assigning, and reviewing[8]. The Issue events API was also considered but ultimately dropped as it contained essentially a subset of the timeline data[9].

GitHub's timeline served as a base but we made several improvements to it for our custom timeline. Because replies to code reviews were missing in GitHub's timeline, we supplemented this data using the PR review comments API[10]. Then, we recovered the original states of dismissed reviews because they were marked as dismissed retroactively. Some actor information was missing in commit events and we supplemented this using the Commits API[11]. GitHub's timeline grouped multiple commit comments, therefore, we expanded them into individual events instead. We then sorted the events by their timestamps to ensure chronology. Finally, we trimmed the events that occurred after the PR closed.

Using GitHub's timeline as a base and making the changes described above allowed us to generate a chronological and near-perfect recollection of any PR.

3.2.3 Selecting factors

A vast amount of PR factors have been studied previously, many of which were found to not influence acceptance (see Table 2.1). While one could study all of them for the sake of completeness, it would be both unfeasible and unnecessary. Furthermore, there are factors yet to be explored. To address these

[7] Docs: `https://docs.github.com/en/rest/issues/timeline#list-timeline-events-for-an-issue`. Last access 2022-11-30.
[8] List of events: `https://docs.github.com/en/developers/webhooks-and-events/events/issue-event-types`. Last access 2022-11-30.
[9] Source: `https://gist.github.com/dahlbyk/229f6ee762e2b0b45f3add7c2459e64a`. Last access 2022-11-30.
[10] Docs: `https://docs.github.com/en/rest/pulls/comments#list-review-comments-on-a-pull-request`. Last access 2022-11-30.
[11] Docs: `https://docs.github.com/en/rest/commits/commits#get-a-commit`. Last access 2022-11-30.

Table 3.3 – Criteria for selecting pull request factors in this work.

Criteria	Description
Changing	The factor must likely change significantly during a PR's lifetime, to facilitate the time-sensitive analysis. Example: *Team_Size* fails this criterion because it is highly constant in a typical PR's lifetime.
Dependent	The factor must not always change naturally independently from the PR, to make sense for our analysis. Example: *Submitter_Age* fails this criterion as it increases over time regardless of the PR.
Distinct	The factor must be somewhat distinct from other selected factors, to avoid overlapping factors. Example: *Discussion_And_Code_Comments* fails this criterion due to being composed of two already selected factors.
Feasible	The factor must be feasible to compute in terms of time and effort, to stay within a manageable scope. Example: *Lines_Changed* fails this criterion since it requires tracking individual changes in the affected files.
Relevant	The factor must not be relevant to only a small subset of data, to ensure applicability to the data. Example: *Is_CSProj_Modified* fails this criterion due to being relevant only to repositories written in C#.

concerns, we designed a set of criteria for selecting factors, which are shown in Table 3.3. As an example, the purpose of the "distinct" criteria was to avoid similar factors to save effort.

We systematically applied these criteria to choose factors from Table 2.1 and to introduce new factors[12]. Table 3.4 presents the final selection of 19 factors subject to the study. A few of the previously studied factors were modified from binary to integral values to increase granularity. For example, *Has_Test_Files* could be "true" for the entirety of a PR which would not produce interesting results, thus we decided to count the number of test files instead. Five novel factors were also introduced, e.g., the number of milestones. For the sake of completeness, Table 3.5 presents the factors that did not make the selection.

[12]This step is not to be confused with feature selection in machine learning. We perform that step later.

Table 3.4 – Pull request factors selected in this work based on the selection criteria. Most factors are from Table 2.1 and some are novel.

Factor	Description
	Previously studied factors
Assignees	# of users assigned to the PR
Build_Fail_Rate	% of failed builds over the total
Changed_Files	# of files touched by the PR
Discussion	# of discussion comments on the PR
Events	# of timeline events on the PR
Labels	# of labels on the PR
Last_Build_Status	0 if the last checked commit had any failing status checks, 1 otherwise
Participants	# of participants on the PR. Section 3.2.1 describes how participants are counted.
PR_Commits	# of commits in the PR
Review_Comments	# of review comments on the PR
	Modified previously studied factors
Conflicts	# of occurrences of the word "conflict" in the PR. Based on Contains_Conflict.
Fixes	# of references to other issues and PRs in the PR. Based on Contains_Fix.
Mentions	# of @-mentions in the PR. Based on Has_Mention_Tag.
Test_Files	# of test files touched by the PR. File names containing "test" are considered test files. Based on Has_Test_Files.
	Novel factors
Approvals	# of reviews in the "approved" state on the PR
Change_Requests	# of reviews in the "changes requested" state on the PR
Cross_References	# of cross-references to the PR from another PR or issue
Intermission	Average time between events on the PR, in minutes
Milestones	# of milestones the PR contributes to

Table 3.5 – Previously studied pull request factors from Table 2.1 that did not meet the selection criteria. All personality and emotion factors did not meet the "feasible" criterion but are omitted from this table for brevity.

Factor	**Criterion not met**	**Factor**	**Criterion not met**
	Factors pertaining to PR changes		
Added_Files	Distinct	Added_Lines	Feasible
Changed_Doc_Files	Relevant	Changed_Other_Files	Relevant
Changed_Source_Files	Relevant	Changes_Config_Or_Settings	Relevant
Classes_Changed	Relevant	Conditionals_Touched	Feasible
Deleted_Files	Distinct	Deleted_Lines	Feasible

Continued on next page

Table 3.5 continued from previous page

Factor	Criterion not met	Factor	Criterion not met
File_Types	Changing	Is_CSProj_Modified	Relevant
Lines_Changed	Feasible	Lines_Changed_Per_Class	Relevant
Loops_Touched	Feasible	Methods_Changed	Feasible
Paths_Changed	Changing	References_Or_Dependencies_Changed	Relevant
Total_Churn	Distinct		
Factors pertaining to PR metadata			
Adds_Feature	Changing	Business_Hours	Changing
Day	Changing	Dependency	Changing
Description_Embedding	Changing	Description_Length	Changing
Has_Merge_Changes	Changing	Has_Stack_Trace	Changing
Hotness	Changing	Is_Deprecating	Changing
Is_Intra_Branch	Changing	Is_Mergeable	Distinct
Is_Refactor	Changing	Last_Commit_Mentions	Distinct
PR_Age	Dependent	Points_To_Issue_Or_PR	Distinct
Queue_Rank	Relevant	Readability	Changing
Release_Cycle_Timing	Changing	Submitted_On_Friday	Changing
Title_And_Description_Length	Changing	Title_Embedding	Changing
Title_Length	Changing		
Factors pertaining to the repository			
Contributors	Dependent	Domain	Changing
External_Commits_Ratio	Changing	Forks	Changing
Lifetime_For_Same_Paths	Changing	Open_Issues	Changing
PR_Commits_Ratio	Changing	Repo_Age	Dependent
Repo_Comments_Per_Merge	Changing	Repo_Comments_Per_PR	Changing
Repo_Files_Changed_Per_PR	Changing	Repo_Language	Changing
Repo_Lines_Added_Rate	Changing	Repo_Lines_Changed_Per_PR	Changing
Repo_Lines_Changed_Weekday_X	Changing	Repo_Lines_Deleted_Rate	Changing
Repo_Merge_Delay	Changing	Repo_Merge_Rate	Changing
Repo_Open_PRs	Changing	Repo_PR_Commits	Changing
Repo_PR_Lifetime	Changing	Repo_PRs	Dependent
Repo_Size	Changing	Repo_Test_Lines_Per_KLOC	Changing
Stars	Changing	Team_Size	Changing
Watchers	Changing		
Factors pertaining to the submitter			
Submitter_Age	Dependent	Submitter_Average_PR_Lifetime	Changing
Submitter_Average_PR_Release_Time	Changing	Submitter_Blobs	Changing
Submitter_Closed_PRs	Changing	Submitter_Closed_Rate	Changing
Submitter_Commit_Comments	Changing	Submitter_Commit_Rate	Changing
Submitter_Commits	Changing	Submitter_Has_Merged_PR	Changing
Submitter_Merge_Rate	Changing	Submitter_Merged_NPM_PRs	Relevant
Submitter_Merged_PRs	Changing	Submitter_NPM_PRs	Relevant
Submitter_Prior_Comments	Changing	Submitter_Project_Contributions	Changing
Submitter_PRs	Changing	Submitter_Repos	Changing
Submitter_Total_Commits	Changing	Submitter_Time_At_Microsoft	Dependent
Submitter_Time_In_Team	Dependent	Submitter_Time_Since_First_Activity	Dependent
Factors pertaining to code review			
Average_Time_Between_Comments	Distinct	Code_Commenting_Reviewers	Distinct
Commenting_Reviewers	Distinct	Discussion_And_Review_Comments	Distinct
Discussion_And_Review_Words	Distinct	First_Human_Response	Changing
Integrator_Availability	Changing	Lifetime_After_First_Response	Changing

Continued on next page

Table 3.5 continued from previous page

Factor	Criterion not met	Factor	Criterion not met
Response_Received	Changing	Submitter_Discussion	Distinct
Wait_Time	Changing		
Factors pertaining to social aspects			
Prior_Interaction	Changing	Reviewer_Follows_Submitter	Changing
Submitter_Affiliation	Relevant	Submitter_Contributions	Changing
Submitter_Events	Changing	Submitter_Followers	Changing
Submitter_Follows	Changing	Submitter_Follows_Closer	Changing
Submitter_Follows_Project	Changing	Submitter_Follows_Reviewer	Changing
Submitter_Issue_Comments	Changing	Submitter_Issue_Events	Changing
Submitter_Issues	Changing	Submitter_PR_Comments	Changing
Submitter_PR_Events	Changing	Submitter_Status	Changing
Factors pertaining to Continuous Integration			
Builds	Dependent	CI_Latency	Changing
CI_Used	Changing	First_Build_Status	Changing
Has_CI_Failure	Distinct		
Factors pertaining to code quality			
Code_Comments_Added	Feasible	Cyclomatic_Complexity_Added	Feasible
Cyclomatic_Complexity_Deleted	Feasible	Has_PMD_Issue_X	Relevant
Has_Test_Code	Distinct	Style_Diff	Feasible
Test_Churn	Feasible		

3.2.4 Collecting data

We collected data in two phases. In the *data selection* phase, the PRs subject to this study were selected with various constraints imposed to ensure relevant and balanced data. Then, in the *data extraction* phase, the previously selected factors (Table 3.4) were extracted from the PRs in the three states of interest (see Section 3.2.1). The following sections elaborate on each phase.

Data selection

We made several decisions on the selection to ensure widely applicable, balanced, and recent data. The first decision on our data was to include the top ten programming languages on GitHub for wide applicability. By number of PRs[13], these were C, C++, C#, Go, Java, JavaScript, PHP, Python, Ruby, and TypeScript. Then, we found repositories that satisfied the following criteria.

[13] Source: `https://madnight.github.io/githut/#/pull_requests/2022/3`. Last access 2022-11-30.

i. The repository is not a fork.

ii. The repository is written in the programming language of interest.

iii. The repository is at least one year old, to ensure a baseline of maturity.

iv. The repository has $\geq$50 contributors, to avoid projects by small teams.

v. The repository has $\geq$200 closed PRs, to ensure the usage of PRs.

GitHub's Search API[14] was used to search and partially filter repositories. The number of contributors and closed PRs were obtained from the HTML of the repository front pages. In each repository, we found PRs that satisfied the following criteria.

i. The PR is closed.

ii. The PR was not created by a bot.

iii. The PR is at most three years old, to ensure recency.

iv. The PR has at least one status check, to ensure the usage of CI.

Again, we used the API for searching and partial filtering. Bots were identified as described in Section 3.2.1 and the presence of status checks was obtained through additional API calls on the PR commits[15].

In our selection, we prioritized the most recently updated repositories and the most recent PRs. We also imposed two limits to avoid imbalance caused by a few dominating languages or repositories. Specifically, exactly 10,000 PRs were selected for each language and repositories were allowed a maximum of 1,000 PRs each. The selection was executed on 2022-07-12 and yielded 100K PRs (10 programming languages $*$ 10K PRs per language) belonging to 239 unique repositories. Some attributes of the repositories are shown in Table 3.6 for additional context.

[14]Docs: `https://docs.github.com/en/search-github/searching-on-github`. Last access 2022-11-30.

[15]Docs: `https://docs.github.com/en/rest/commits/statuses#get-the-combined-status-for-a-specific-reference`. Last access 2022-11-30.

Table 3.6 – Attributes of the repositories in the data selection. Gathered on 2022-10-17.

Repository attribute	Min	Median	Max	Mean
Age (years)	2	6	14	6.6
Closed pulls	254	3K	108K	6K
Commits	336	9K	650K	22K
Contributors	50	169	5K	333.6
Forks	5	640	51K	2K
Size (MB)	1	111	20K	392.2
Stars	2	3K	93K	7K

Data extraction

Extracting the factors of a PR began by generating its timeline as described in Section 3.2.2. Then, the indices of the ready, middle, and closure states were located on the timeline as defined in Section 3.2.1. We generated each state by stepping through the timeline up to the corresponding index, e.g., generating the closure state meant stepping through the entire timeline. While stepping, we updated the factors of interest according to the current timeline event, e.g., a "labeled" event would increment *Labels*. See Section 1.4 for full technical detail on what API endpoints were used, how timelines were generated, and how factors were computed.

We computed the factors in each state for all of the 100K selected PRs. The process yielded 5.7M factor values (100K PRs $*$ 19 factors $*$ 3 states). The majority (85%) of the PRs were merged. Statistics on the extracted factors are shown in Table 3.7 for additional context.

3.2.5 Modeling acceptance

PR acceptance may be modeled as a *binary classification problem*. A generic description of classification is simply to label which class a sample belongs to based on its features. In terms of PR acceptance, the label is the merge status, the classes are {*merged, non-merged*}, the samples are PRs, and the features are factors. Using more specific terms, our task is to predict whether a PR was merged or not based on its factors. The following sections detail how we executed this task to answer RQ1 and RQ2.

Table 3.7 – Statistics on the pull request factors pre-transformation.

Factor	Min	25%	Median	75%	Max	Mean
Approvals	0	0	0	1	26	0.5
Assignees	0	0	0	0	7	0.2
Build_Fail_Rate	0.0	0.0	0.0	0.2	1.0	0.2
Change_Requests	0	0	0	0	24	0.1
Changed_Files	0	1	2	5	3248	11.6
Conflicts	0	0	0	0	315	0.1
Cross_References	0	0	0	0	28	0.2
Discussion	0	0	0	2	507	1.3
Events	1	4	7	14	3945	12.8
Fixes	0	0	0	1	898	1.1
Intermission	0	0	21	231	891298	768.2
Labels	0	0	0	1	88	0.9
Last_Build_Status	0	1	1	1	1	0.8
Mentions	0	0	0	0	1721	0.7
Milestones	0	0	0	0	4	0.1
Participants	0	0	1	2	88	1.1
PR_Commits	0	1	1	2	250	3.1
Review_Comments	0	0	0	1	339	1.4
Test_Files	0	0	0	1	2378	2.4

Feature preprocessing

We applied *feature transformation* and *feature selection* to prepare our data for model training to improve the results. Similar preparations have been used previously [4, 1, 14, 22]. For context, our initial set of features and descriptive statistics may be seen in Table 3.7.

We log-transformed (i.e., $log(x)$ or $log(x+0.5)$ if necessary) and standardized[16] ($\overline{x} = 0$, $\sigma = 1$) our features to minimize skewness and de-emphasize outliers. This was applied to the ready, middle, and closure data separately. Then, we paired and tested all features for high correlation. A pair was highly correlated if the *Spearman's rank correlation coefficient* met or exceeded a threshold ($|\rho| \geq 0.7$) with statistical significance ($p \leq 0.05$)[17]. *Approvals* and *Review_Comments* was the only highly correlated pair. Instead of arbitrarily choosing which feature to drop, we used the following strategy. First, we calculated the correlations between *Approvals* and all other factors, and we did

[16]Function used: `https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html`. Last access 2022-11-30.

[17]Function used: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.spearmanr.html`. Last access 2022-11-30.

the same for *Review_Comments*. Then, we removed the feature with the highest mean correlation to reduce overall multicollinearity. *Review_Comments* was dropped in this step. We measured correlation only in the closure data to imitate the typical use case of only having data from closed PRs available.

Model training

Our models were trained using *10-fold cross-validation*, which has been used in the past [9, 8, 17]. The strategy begins by shuffling the experiment data and dividing it into 10 equal-sized folds (partitions). Then, the model is trained on 9 folds and the remaining fold is used for generating predictions. Training occurs 10 times such that every fold is used for testing exactly once. Assessing the prediction quality is a matter of averaging the scores from each training cycle. 10-fold is robust against variance that may arise due to sampling a single training and testing set. Moreover, we chose stratified folds, meaning that they maintained the same class distribution of merged to non-merged PRs as the entire data set.

We chose algorithms for the experiments based on their prevalence in the related works [4, 12, 8]. We selected AdaBoost, Bagging, Decision Tree, Gaussian Naive Bayes, Linear SVM, Logistic Regression, and Random Forest. All algorithms were available in the Python package Scikit-Learn[18]. We increased the maximum number of iterations for Logistic Regression and Linear SVM to reach convergence but did not perform any additional tuning. Each algorithm was trained on the closure data and tested on the ready, middle, and closure data. Only the closure data was used for training to mirror the typical situation where only closed PRs are used for training.

Performance metrics

Measuring the performance of a classifier starts by feeding it with test features and comparing the predictions to the test labels. In this case, the classifier is fed PR data it has not seen yet, predicts the merge statuses, and the prediction is compared to reality. The results of such a comparison serve as the foundation for calculating performance metrics, of which many expose different aspects of a prediction. Table 3.8 lists the metrics in this work based on their appearances in the related works [18, 4, 12]. We used these metrics to measure the performance difference between predicting on the ready, middle, and closure data to address RQ1.

[18] Available at `https://scikit-learn.org/`. Last access 2022-11-30.

Table 3.8 – Performance metrics for evaluating the classifiers.

Metric	Class	Explanation
Accuracy (ACC)	Both	Rate of correct predictions among all predictions.
Precision (PREC)	Merged	Rate of correct merge predictions among all merge predictions.
	Non-merged	Rate of correct non-merge predictions among all non-merge predictions.
Recall (REC)	Merged	Rate of correct merge predictions among all actual merges.
	Non-merged	Rate of correct non-merge predictions among all actual non-merges.
F_1-score (F_1)	Merged	Harmonic mean between merge precision and merge recall.
	Non-merged	Harmonic mean between non-merge precision and non-merge recall.
Area Under Curve (AUC)	Both	Ability to distinguish merges from non-merges. Ranges from 0 to 1. 0.5 indicates a random prediction.
Matthews Correlation Coefficient (MCC)	Both	Correlation coefficient that effectively captures all aspects of a binary classification at once. Ranges from -1 to 1. 0 indicates a random prediction.

Feature importance

The influence of a feature on a particular outcome is known as feature importance. Three common measurements are *Gini importance* (or Mean Decrease in Impurity), *permutation importance* (or Mean Decrease in Accuracy), and *drop-column importance*. In the latter, importance is obtained by retraining the model with one feature dropped and measuring the performance change against the baseline. For example, if a model achieves an accuracy of 0.80 with all features and 0.78 without one particular feature, the importance of said feature is 0.02. The main benefit is its resistance to bias unlike the other two approaches and the main drawback is time because each measurement requires retraining the model [28]. We opted for drop-column importance because retraining was relatively fast. Furthermore, we chose accuracy as the metric for measuring performance change because it is intuitive and has been used in a related work that adopted this approach [12].

We used the model that predicted with the highest overall accuracy to measure importance since that was our metric of choice. With the model, we measured the drop-column importance of each feature in each state to address RQ2. A total of 57 measurements were made (19 features $*$ 3 states).

3.3 Qualitatively studying pull request user actions and opinions (RQ3 and RQ4)

In this section, we describe the survey conducted by creating and releasing a questionnaire to answer RQ3 and RQ4. The questionnaire inquired about the actions and opinions of GitHub users concerning PR development and review. In Section 3.3.1, we outline the preliminary decisions made before creating the questionnaire. We explain the questionnaire design and its refinement in Section 3.3.2 and Section 3.3.3, respectively. In Section 3.3.4, we specify the task of generating and sending invitations. Then, in Section 3.3.5, we detail how undesirable responses were detected and partially or completely discounted. Finally, we describe the coding approach we employed to analyze the free-text responses in Section 3.3.6. The questionnaire and its raw and coded responses are available in Section 1.4.

3.3.1 Preliminaries

We opted for a questionnaire as the means of executing the survey, a recurring approach in previous related works [17, 3, 8, 1, 6, 2]. Our questions were mainly multiple-choice for the respondent's convenience and ease of analyzing results. The multiple-choice options were based on previous questionnaires and feedback as described in Section 3.3.3. Google Forms was decided as the form creator because it offered the necessary features and has been used previously [17, 1]. Our questionnaire was anonymous to increase participation and anyone with the link could answer because it has boosted responses in the past [2, 10]. Lastly, we targeted GitHub users of any experience level who have reviewed or submitted PRs in the last year, to ensure broad applicability and relevancy of the results.

We created one questionnaire for both roles instead of having separate ones for submitters and reviewers. There were multiple reasons for this. First, respondents who have both roles would not have to answer the demographic questions twice. Second, having one questionnaire creates an additional group for analysis, i.e., persons who have both roles, which would be impossible with separate anonymous questionnaires. Third, having one questionnaire is simply more convenient to share.

3.3.2 Questionnaire design

We grouped the questionnaire into four sections to allow submitters to easily skip reviewer questions and vice versa. The first section was an introductory text which described the target group, context, and goal of the survey. It also thanked the reader and encouraged them to share it.

The subsequent two sections, which covered the submitter and reviewer questions, are best described simultaneously due to their similarities and deliberate synergy. Both sections were optional and first asked for the respondent's submission or review activity in the last year in a multiple-choice question. Then, several questions were asked about the respondent's actions and opinions concerning the development and review of PRs. Specifically, the following questions were asked[19].

Q1 (for submitters) *When preparing a pull request for its first review, how often do you use each action to increase your chances of having it merged?*

Q2 (for reviewers) *When giving a pull request its first review, how influential is each aspect for your impression?*

Q3 (for reviewers) *How often do you use each action to improve the chances of a pull request being merged?*

Q4 (for submitters) *When responding to feedback on your pull request, how often do you use each action to increase your chances of having it merged?*

Q5 (for reviewers) *How influential is each aspect when making your final decision on a pull request?*

Q1-5 were meant to depict the interplay between submitters and reviewers from PR creation to closure (see Figure 2.1) while addressing RQ3 and RQ4. They were all multiple-choice grids with five columns asking the respondent to rank the frequency or influence of the relevant item. Questions on frequency ranged from "Very often" to "Rarely" while questions on influence ranged from "Very important" to "Insignificant". For example, Q1 included the item "I follow the submission guidelines of this repository" and Q2 included "Clear language in the title/description". There was a "Not sure" column and free-text questions for missing options.

[19] The numbering and order of the questions are only for illustrative purposes. In the questionnaire, Q1 and Q4 were in the submitter section, and Q2, Q3, and Q5 were in the reviewer section.

The final questionnaire section, which contains the demographic questions, was meant to obtain context to responses. Specifically, it inquired about the focus of the respondent's career, their experience, domain, degree, and whether they have researched PRs on GitHub. All questions were mandatory because they could be answered by all respondents, unlike the submitter and reviewer questions. This was crucial for comparing groups of respondents and understanding the limitations in terms of the applicability of results.

3.3.3 Refinement process

The questionnaire went through several iterations before public release based on three rounds of feedback coming from multiple parties. The feedback generally regarded missing multiple-choice options and ambiguous language. All parties were told the research questions targeted by the questionnaire as well as the overarching goal of the thesis. The first round was performed by the project student and supervisor together and the examiner reviewed jointly in the second round. In the third and final round, three people with no knowledge of this work provided feedback via email[20]. These contacts were peers of and recommended by the supervisor and examiner.

3.3.4 Sending invitations

We invited two groups to the questionnaire via email: GitHub users and authors of the related works. We also sent the questionnaire to the people who gave feedback during its development but they were asked to only share and not respond to it due to bias. Invitations were sent on 2022-04-13 and 204 responses were collected, the last being received on 2022-05-10 (28 days elapsed). The remainder of this section details how we generated and sent personalized invitations.

Inviting GitHub users

We sent the vast majority of our invitations to GitHub users. Considering the response rates in the related works [1, 3], we anticipated that 2,000 invites would yield a reasonable number of responses. Users were found using the following process. First, we used GitHub's Search API[21] to find the latest PRs

[20] Thank you Jean-Rémy Falleri, Simon Larsén, and the third critic who did not wish to include their name.

[21] Docs: `https://docs.github.com/en/rest/reference/search#search-issues-and-pull-requests`. Last access 2022-11-30.

in the relevant repositories. We only considered the repositories in our data selection (see Section 3.2.4) because they were established and had a history of using PRs. Then, we stored the account details of users who had submitted, commented on, merged, or closed a PR. Bot accounts were detected using the method in Section 3.2.1 and discarded. The actions a user had taken on a PR were retrieved from the PR's timeline (see Section 3.2.2).

With the account details on hand, we generated invitations based on a template and personalized them with the receiver's username, or full name if available, and the repository they were found in. The template gave context on this work, the questionnaire, and key points such as completion time and anonymity. It also encouraged sharing and linked the questionnaire at the bottom. Receivers were offered to be contacted when the work was complete which we expected would increase participation. We emailed the 2,000 invitations in an automated fashion.

Inviting authors of related works

In addition to GitHub users, we invited 68 authors of related works. We expected that this group would yield a higher response rate than the other one due to belonging to the research community and having a passion for the same broader topic. The emails were based on a template, personalized again by the receiver's full name and also their related paper. The invitations looked similar to the ones for GitHub users except that we expressed gratitude to the author for their valuable contribution referenced in this work. We added a demographic question to quantify the responses from this group.

3.3.5 Filtering responses

Open and anonymous surveys come with an inherent risk of undesirable (but sometimes entertaining) responses, commonly due to confusion or mischief, hence the need for filtering responses. First, the project student and supervisor judged the apparent seriousness of each free-text response. The entire response was dropped if both parties agreed that it was unserious beyond a reasonable doubt. Two responses were dropped entirely in this step. Second, responses claiming no recent review or submission activity had their corresponding portion of the response removed, for those answers were irrelevant. 12 responses had one section removed in this step. We did not perform additional filtering to reduce potential bias.

3.3.6 Analyzing free-text responses

After filtering our responses, we applied open coding to the free-text answers, an approach used by related works [3, 6]. Respondents could freely write in six instances: the demographic question asking for their domain included an "Other" option and Q1-5 had an associated question each asking for missing options (see Section 3.3.2 for info on Q1-5). The free-text responses were labeled by the project student and then the labeling was reviewed by the supervisor in live sessions with the student. All disagreements were resolved such that both parties agreed with the labeling.

During coding, we observed that the free-text questions related to Q1-Q5 were partly used to elaborate on the respondent's previous answers, unlike the original intention of only supplementing multiple-choice options. We ignored these "elaborative" parts of the responses when labeling and used them for additional insights when discussing the results.

Chapter 4

Study results

In this chapter, we use the results obtained through executing the study design to answer the research questions. The prediction results (RQ1 and RQ2) are presented in Section 4.1 and the survey results (RQ3 and RQ4) are presented in Section 4.2.

4.1 Prediction results

This section showcases the results from the machine learning experiments described in Section 3.2. Models were trained on data from closed PRs to predict acceptance, i.e., the merging outcome. They were then tested on PR data from three distinct states called ready (ready for review), middle (halfway complete), and closure (closed). In Section 4.1.1, we present the performance changes when making predictions on PRs in the aforementioned states. Then, in Section 4.1.2 we present the importance of each feature for the predictions in each state.

4.1.1 How does the prediction of acceptance change when predicting on open pull requests? (RQ1)

To answer RQ1, we trained classifiers on closure data and tested them on ready, middle, and closure data to observe performance changes when predicting the acceptance of PRs in different states. Table 4.1 shows the raw performance results of our model predictions (see Table 3.8 for info on the metrics). Moreover, Table 4.2 summarizes the performance change between predicting on the different states.

Table 4.1 – Performance of the pull request acceptance classifiers. From top to bottom, each cell contains the score when testing on the ready, middle, and closure data, as well as the mean inside parenthesis. Highest mean of each metric in bold. Single asterisk implies same score compared to when testing data from the previous state, double asterisks imply worse score.

Metric	Class	AdaBoost	Bagging	Decision Tree	Gaussian Naive Bayes	Linear SVM	Logistic Regression	Random Forest
ACC	Both	0.84	0.71	0.70	0.77	0.84	0.84	0.78
		0.82**	0.77	0.75	0.80	0.84*	0.84*	0.81
		0.89	0.91	0.88	0.84	0.87	0.87	0.92
		(0.85)	(0.80)	(0.78)	(0.80)	**(0.85)**	**(0.85)**	(0.83)
PREC	Merged	0.85	0.88	0.88	0.87	0.85	0.85	0.87
		0.88	0.91	0.90	0.89	0.86	0.86	0.91
		0.90	0.94	0.93	0.90	0.87	0.88	0.93
		(0.88)	**(0.91)**	(0.90)	(0.88)	(0.86)	(0.86)	(0.90)
	Non-merged	0.30	0.24	0.23	0.24	0.26	0.24	0.27
		0.38	0.34	0.31	0.35	0.41	0.40	0.39
		0.72	0.71	0.59	0.46	0.75	0.68	0.79
		(0.47)	(0.43)	(0.38)	(0.35)	(0.47)	(0.44)	**(0.48)**
REC	Merged	0.99	0.77	0.74	0.86	0.99	0.99	0.87
		0.92**	0.81	0.80	0.88	0.98**	0.97**	0.87*
		0.97	0.95	0.93	0.91	0.99	0.98	0.97
		(0.96)	(0.84)	(0.82)	(0.88)	**(0.99)**	(0.98)	(0.90)
	Non-merged	0.02	0.41	0.43	0.25	0.01	0.01	0.27
		0.29	0.55	0.52	0.36	0.08	0.12	0.49
		0.39	0.64	0.60	0.44	0.18	0.28	0.62
		(0.23)	**(0.53)**	(0.52)	(0.35)	(0.09)	(0.14)	(0.46)
F_1	Merged	0.92	0.82	0.81	0.86	0.92	0.92	0.87
		0.90**	0.86	0.85	0.88	0.91**	0.91**	0.88
		0.94	0.95	0.93	0.90	0.93	0.93	0.95
		(0.92)	(0.87)	(0.86)	(0.88)	**(0.92)**	**(0.92)**	(0.90)
	Non-merged	0.04	0.30	0.30	0.24	0.03	0.03	0.27
		0.32	0.42	0.39	0.36	0.13	0.18	0.44
		0.51	0.67	0.60	0.45	0.30	0.39	0.69
		(0.29)	**(0.47)**	(0.43)	(0.35)	(0.15)	(0.20)	**(0.47)**
AUC	Both	0.51	0.59	0.59	0.56	0.50	0.50	0.57
		0.60	0.68	0.66	0.62	0.53	0.54	0.68
		0.68	0.80	0.76	0.67	0.59	0.63	0.79
		(0.60)	**(0.69)**	(0.67)	(0.62)	(0.54)	(0.56)	(0.68)
MCC	Both	0.04	0.15	0.14	0.11	0.03	0.02	0.14
		0.23	0.30	0.26	0.24	0.12	0.15	0.33
		0.48	0.62	0.52	0.35	0.33	0.38	0.65
		(0.25)	(0.36)	(0.31)	(0.23)	(0.16)	(0.18)	**(0.37)**

Across 126 score differences (9 metrics $*$ 7 algorithms $*$ 2 state changes), performance did not change in three cases and decreased in seven. All decreases except for one were small ($\leq$0.02). In fact, all metrics improved on average across all the models save for the merge recall. The non-merged metrics were remarkably improved, which makes sense because the merged metrics tended to take on higher scores. The second highest improvement was to the

Table 4.2 – Average performance change to the classifiers between testing data from one pull request state to another. For example, the top-left value indicates that the models gained 0.02 accuracy on average when testing on middle data compared to ready data.

State change	ACC	PREC (Merged)	PREC (Non-merged)	REC (Merged)	REC (Non-merged)	F_1 (Merged)	F_1 (Non-merged)	AUC	MCC
Ready $\longrightarrow$ middle	+0.02	+0.02	+0.11	0.00	+0.14	+0.01	+0.15	+0.07	+0.14
Middle $\longrightarrow$ closure	+0.08	+0.02	+0.30	+0.07	+0.11	+0.05	+0.20	+0.09	+0.24

MCC score, indicating that predictions tended to drastically improve holistically. These observations suggest that training on closed PRs while predicting on open PRs almost always leads to a significant performance decrease. The mean improvement to all metrics was 0.07 from the ready to the middle state and 0.13 from the middle to the closure state. This indicates that as PRs near the end of their lifetime, models trained on closed PRs become exponentially more successful at predicting the merging outcome.

When judging the absolute scores, as opposed to relative changes, the class distribution of 85% merged PRs and 15% non-merged should be considered. For instance, a naive model that always predicts a merge would achieve an accuracy of 0.85, or 0.15 if it always predicted non-merges. A random model would yield 0.50 in the AUC score and 0.00 in MCC. With this in mind, the minimum accuracy achieved was not particularly impressive (0.70). The AUC scores ranged from random to high (0.50–0.80) and the MCC ranged from near random to moderate (0.02–0.65). This tells us that our models were equivalent to or worse than a naive model in some metrics when predicting PRs in the ready state. All class-specific metrics achieved high scores in the merged class (0.74–0.99) while the non-merged scores were poorer overall and far more varied (0.01–0.79).

Key takeaways for RQ1. Training on closed PRs while predicting the acceptance of open PRs almost always lead to significant performance loss. Our models were even less successful in some metrics than naive models when predicting PRs in the earliest state. However, our models improved exponentially when tested against PRs nearer the completed state.

4.1.2 How important is each factor when predicting the acceptance of pull requests across their lifetime? (RQ2)

To answer RQ2, we measured the importance of each factor when training on closure data and predicting acceptance on ready, middle, and closure data. Linear Regression was our model of choice as it achieved the highest mean accuracy in the previous experiments. Moreover, we used the drop-column strategy, i.e., the model was retrained and retested without each feature, and then the accuracy change was measured as the importance value (see Section 3.2.5 for an elaboration).

Table 4.3 presents the feature importance measurements ordered by descending importance. For example, the importance of the *Last_Build_Status* in the closure state was +0.00915, meaning that said feature contributed that quantity of accuracy. On the other hand, the importance of the number of *Change_Requests* in the middle state was -0.00167, meaning that we gained some accuracy by removing said feature from the model. While many of the negative importance values are small and within random variance, the larger values are not typically seen in modeling. The explanation is that our model was misled when training on closed PRs and tested on PRs in open states. This makes sense considering our previously shown accuracy losses when predicting on open PRs (see Table 4.1).

It is interesting how the most important closure features are shared across the other states. Specifically, the features are the number of code review *Approvals*, the *Last_Build_Status*, and the amount of *Discussion*. These are all important features in the open states, but also in the reverse direction, e.g., *Approvals* oscillates between the top and bottom ranks across the states. Perhaps these features were particularly valuable in the closure state, therefore, the model was prone to over or underestimate their value in the open states. The takeaway is that some features can either be substantially helpful or harmful depending on the current state of the PR. For example, modeling the *Discussion* and *Last_Build_Status* could hurt the prediction of PRs that are ready to review but help the prediction of PRs in later states.

Finally, two features were consistently neutral or substantially helpful in all states. The number of *Events* was negligible in the ready state but was valuable to the prediction in the other states. The number of *Participants* was important in the ready state but negligible in the rest. This suggests that some features can be modeled without any downside despite training on closed PRs and predicting on PRs in any state.

Table 4.3 – Feature importance for pull request acceptance in the three states. The drop-column strategy was used with the accuracy metric and a Logistic Regression model.

Ready	Change	Middle	Change	Closure	Change
Approvals	+0.00943	Events	+0.00261	Approvals	+0.01095
Participants	+0.00144	Last_Build_Status	+0.00217	Last_Build_Status	+0.00915
Assignees	+0.00036	Discussion	+0.00198	Discussion	+0.00369
Cross_References	+0.00006	Labels	+0.00029	Events	+0.00156
Conflicts	-0.00002	Changed_Files	+0.00004	PR_Commits	+0.00081
Fixes	-0.00004	Assignees	-0.00005	Change_Requests	+0.00080
Labels	-0.00007	Conflicts	-0.00010	Labels	+0.00079
Changed_Files	-0.00013	PR_Commits	-0.00013	Assignees	+0.00050
Milestones	-0.00016	Milestones	-0.00014	Cross_References	+0.00046
Build_Fail_Rate	-0.00018	Mentions	-0.00016	Test_Files	+0.00040
Change_Requests	-0.00037	Cross_References	-0.00017	Participants	+0.00038
Mentions	-0.00037	Test_Files	-0.00018	Changed_Files	+0.00033
Test_Files	-0.00044	Fixes	-0.00027	Mentions	+0.00028
Events	-0.00078	Build_Fail_Rate	-0.00029	Milestones	+0.00025
PR_Commits	-0.00098	Participants	-0.00038	Conflicts	+0.00012
Intermission	-0.00109	Change_Requests	-0.00167	Fixes	0.00000
Last_Build_Status	-0.00228	Intermission	-0.00301	Build_Fail_Rate	-0.00007
Discussion	-0.00444	Approvals	-0.01143	Intermission	-0.00089

Key takeaways for RQ2. The top features in the closure state were the number of approved code reviews, the last build status, and the amount of discussion. These were also important features in the open states, but their effect fluctuated between helpful and harmful, i.e., they were sensitive to the current state of the PR. Other features were either neutral or positive across all states, meaning that they could be modeled with no downside.

4.2 Survey results

This section showcases the results from our survey described in Section 3.3. Specifically, we present how submitters prepare PRs and respond to feedback in Section 4.2.1 and how reviewers respond to PRs and judge them in the first and final review in Section 4.2.2. Our survey received 202 valid responses, of which 196 were on the submitter section and 171 were on the reviewer section, yielding a participation rate of 9.9% (204 total responses/2,068 invites). We have summarized the demographic information below for context on the results.

Table 4.4 and Figure 4.1 showcase the work experience and domains of the respondents. Most respondents had five or more years of work experience (73%) and the most prevalent domains were software engineering (35%) and web (22%). When asked about the current focus of their software engineering career, 73% (145) answered industry, 8% (17) academia, 14% (29) both, and 5% (11) none. In terms of computer science education, 34% (69) had a bachelor's degree, 31% (63) had none, 29% (59) had a master's, and 6% (11) had a doctorate. Finally, 26% (52) had researched the topic of PRs on GitHub. In summary, the respondents typically had five or more years of work experience, worked in software engineering or web, focused on industry, had an even distribution of education levels below doctorate, and had not researched PRs on GitHub.

Table 4.4 – Work experience in the computer science field of the survey respondents.

# of years	Responses (202)
<1	9% (19)
1–4	18% (36)
5–9	31% (62)
≥10	42% (85)

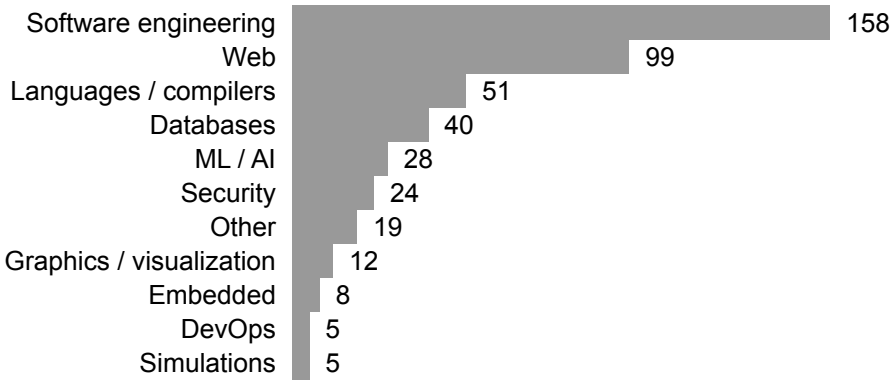

Figure 4.1 – Computer science domain of the survey respondents. Domains with less than five responses are omitted. Respondents could select multiple domains.

4.2.1 How do submitters prepare pull requests and respond to feedback to increase acceptance? (RQ3)

To answer RQ3, we examined the answers to the submitter section of our survey. This section inquired about the respondent's submission activity and actions when preparing PRs and responding to feedback to increase acceptance. Table 4.5 displays the activity levels of the 196 respondents. The most common activity level was more than 50 PR submissions in the last year. We have summarized the results of the remaining results below.

Table 4.5 – Pull request submission activity of the survey respondents.

# of submissions in the last year	Responses (196)
1–10 pull requests	27% (52)
11–30 pull requests	23% (45)
31–50 pull requests	14% (28)
>50 pull requests	36% (71)

Actions when preparing pull requests

We asked submitters how they prepare PRs to increase acceptance in a multiple-choice question. Figure 4.2 shows the responses in terms of how often they use each action. The ten most common actions are comfortably on the left-hand side of the figure, receiving at least 69% "very often" and "often" votes. This suggests that submitters usually employ many methods to prepare their PRs. Clarity was the top priority, receiving only one "occasionally" vote and zero "rarely" votes. Three common actions were related to the relevance of the submission, i.e., referencing an issue, communicating the value of the PR, and checking for competing PRs. The remaining six actions all pertained to project fitness. In summary, submitters prioritize clarity, project fitness, and relevance when preparing PRs.

Respondents could supplement their multiple-choice answers in a subsequent free-text question. We received 21 such responses, 15 of which suggested new actions and the rest elaborated on their previous answer. The most common suggestion was making sure the changes are relevant with five responses, reiterating the importance of relevance. Three actions received two responses each. These were changing the PR commit history, creating or mak-

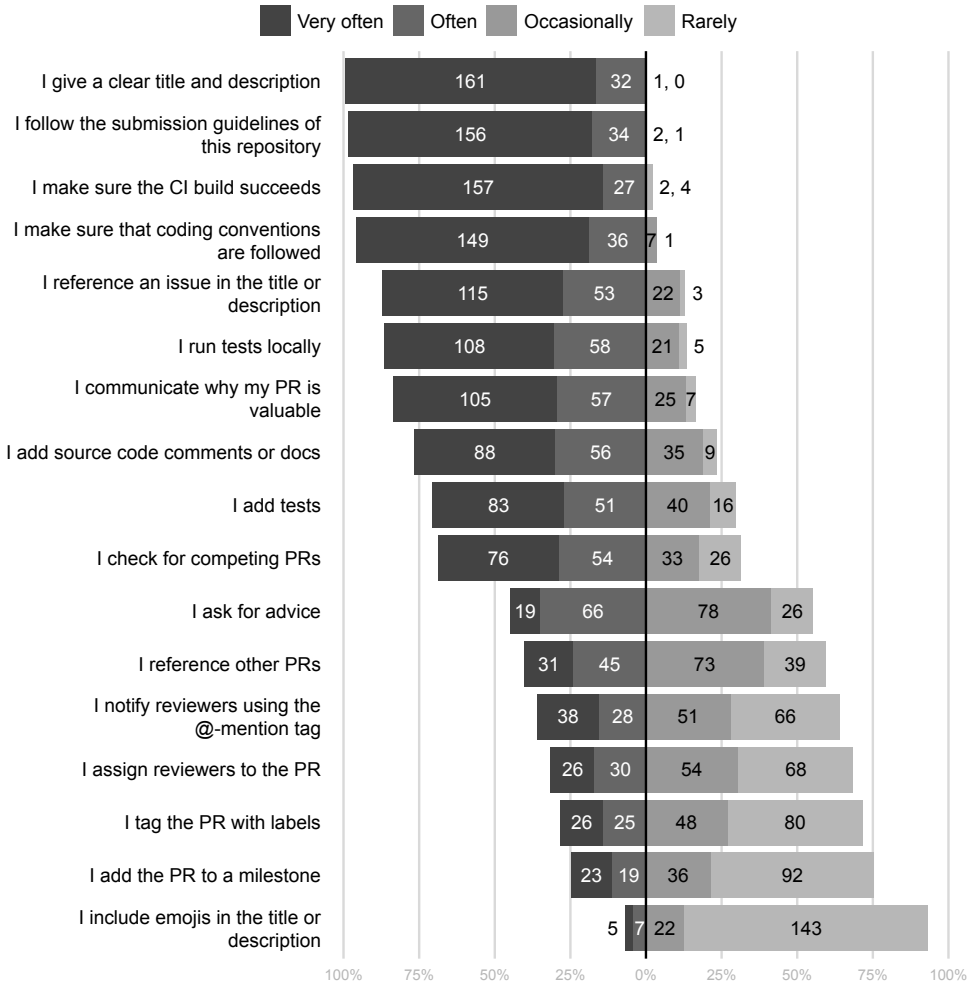

Figure 4.2 – Actions performed by submitters when preparing pull requests to increase acceptance, ordered by frequency. Some actions are rephrased for brevity.

ing sure an issue exists, and checking the repository's history of responding to PRs. The two former actions reinforce previous findings while the latter was unique. One respondent mentioned using the latter action very often and implied they would not even aspire to create PRs in repositories that are too inactive on PRs. Finally, two responses mentioned marking their unprepared PRs as works in progress.

Actions when responding to feedback

In another multiple-choice question, we asked submitters how they respond to feedback on their PRs to increase acceptance. Figure 4.3 shows the responses in terms of how often they use each action. The most frequent actions were responding quickly, fixing flaws, clarifying intent, and increasing compliance. The least frequent actions were asking for advice, asking the reviewer to elaborate, assuring the reviewer that nothing is wrong, and describing why the PR is preferable over competing ones. In summary, submitters try to respond quickly, usually by adjusting their PR or clarifying their intent, they tend to understand the feedback, and know how to proceed.

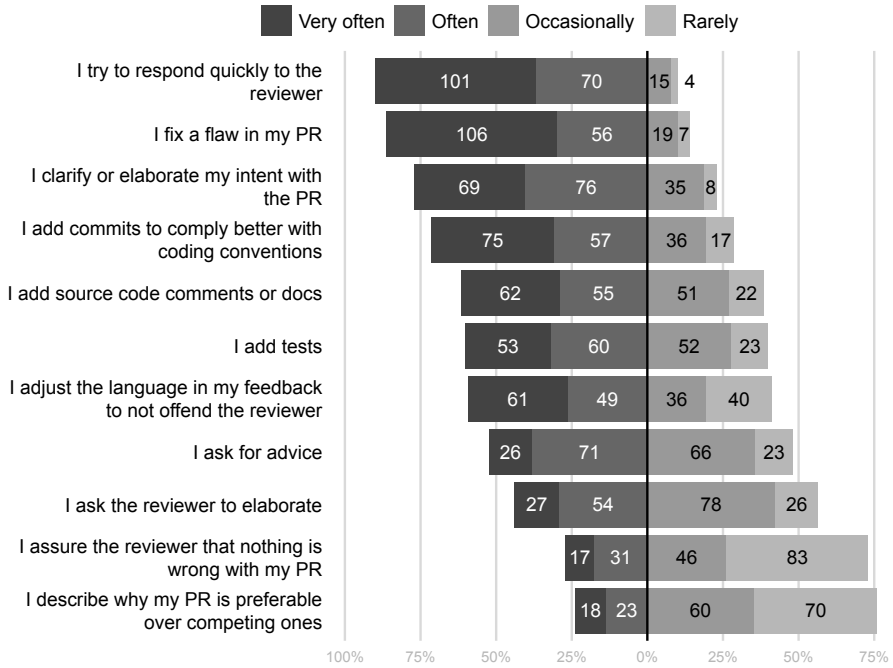

Figure 4.3 – Actions performed by submitters when responding to pull request feedback to increase acceptance, ordered by frequency. Some actions are rephrased for brevity.

We also gave respondents a supplementary free-text question that received 13 responses, eight of which suggested new actions. Changing the PR commit history was suggested five times, a large amount relative to the other free-text results, and adjusting the PR to address a specific comment was suggested twice. The actions are related to previous findings that submitters tend to focus on project fitness and adjusting their PRs.

Key takeaways for RQ3. Submitters prepare their PRs by emphasizing project fitness, relevance, and clarity. They try to respond quickly to feedback, which usually means adjusting their PR or clarifying their intent. Moreover, submitters tend to understand the feedback they receive and know how to proceed.

4.2.2 How do reviewers judge and respond to pull requests to increase acceptance? (RQ4)

To answer RQ4, we examined the answers to the reviewer section of our survey. This section inquired about the respondent's review activity, opinions, and actions when judging and giving feedback on PRs to increase acceptance. Table 4.6 displays the activity levels of the 171 respondents. The most common activity level was more than 50 PR reviews in the last year. The answers to the remaining questions are summarized below.

Table 4.6 – Pull request review activity of the survey respondents.

# of reviews in the last year	Responses (171)
1–10 pull requests	25% (43)
11–30 pull requests	19% (33)
31–50 pull requests	11% (19)
>50 pull requests	45% (76)

Influences on pull request review

We asked reviewers how they judge PRs in the first review and final decision to increase acceptance. They were tasked with rating the influence of several aspects in two multiple-choice grids, one for each review. Figure 4.4 displays the responses ordered by influence. The "score" columns refer to the *influence score*, a weighted sum that quantifies the degree of influence of each aspect. For instance, code complexity received 51 "very important", 80 "important", 28 "matters somewhat", and 6 "insignificant" responses, yielding the score 341 ($3 * 51 + 2 * 80 + 1 * 28 + 0 * 6$).

Note that the aspects are ordered based on average score, i.e., the top and bottom aspects are the most and least influential *overall*. Sorting them in the first review separately and doing the same in the final decision provides us with deeper insight. In the first review, the five most influential aspects are

3 - Very important 2 - Important 1 - Matters somewhat 0 - Insignificant	First review				Score	Final decision				Score
	3	2	1	0		3	2	1	0	
Status of the most recent CI build	86	43	22	14	366	119	28	5	4	418
Compliance to repository standards	78	62	26	4	384	92	49	16	2	390
Whether there is a conflict	55	37	51	22	290	104	33	11	11	389
Presence of tests in the pull request	66	49	41	13	337	71	49	30	8	341
States of the code reviews by other users						67	54	29	9	338
Code complexity	51	80	28	6	341	49	63	37	8	310
Clear language in the title or description	44	74	42	8	322	36	62	45	17	277
Presence of source code comments or docs	34	60	58	16	280	41	55	45	17	278
Size (# of commits, changed files, LOC)	48	59	41	21	303	31	49	48	29	239
Clear language in the commit messages	32	65	52	20	278	36	49	46	30	252
Attitude expressed by the submitter	32	57	63	13	273	13	48	67	28	202
Responsiveness of the submitter						19	67	46	25	237
Whether a specific GitHub issue is fixed	28	48	52	36	232	21	44	51	43	202
Whether a feature is added	24	51	40	45	214	16	43	43	52	177
Your past interaction with the submitter	16	31	65	54	175	20	20	63	65	163
History of the submitter's previous PRs	14	34	58	56	168	5	25	57	65	122
Amount of discussion in the thread						10	26	50	68	132
Status of the submitter in this repository	11	31	53	62	148	7	22	48	77	113
Whether a milestone is contributed towards	9	27	47	76	128	9	25	44	76	121
Number of code review comments						6	29	48	74	124
Presence of labels	4	14	38	95	78	9	15	35	96	92

Figure 4.4 – Aspects influencing the first and final review of a pull request. "Score" refers to the *influence score*, a weighted sum defined in Section 4.2.2. Ordered by descending mean score. Some aspects were not asked for the first review, hence the omission of some cells.

compliance (384 influence score), CI status (366), code complexity (341), test inclusion (337), and a clear title or description (322), respectively. The top ranking in the final decision is similar but code complexity and a clear title or description drop out while the presence of conflict and states of reviews by other users appear. These findings suggest that reviewers value project fitness consistently, especially in the final decision, and they also appreciate simplicity and clarity in the first review.

We can observe how aspects vary in influence by measuring the change to the score between the first and final review. The presence of conflict experienced the largest increase in influence by far, followed by CI status. PR size, the submitter's attitude, the submitter's PR history, and a clear title or description all experienced the largest decrease. The influence scores decreased in the final decision on average, but despite this, those related to project fitness only increased. These findings reinforce the previous finding that reviewers prioritize project fitness above all. Additionally, reviewers value other aspects less overall in the final decision.

We can estimate the consensus between responses by taking the standard deviation of the number of responses in each option. For example, an aspect with similar amounts of responses in each option implies weak agreement and a low standard deviation. Four aspects were among the five least agreed-upon aspects in both reviews. These were PR size, clear commit messages, whether a feature is added, and whether a GitHub issue is fixed. Furthermore, the presence of conflict rose from the third lowest agreement to the second highest, the largest change by far. We interpret aspects with weak consensus as more context-sensitive. Two respondents alluded to this, one claiming they do not use milestones and the other claiming that some repositories require an issue to be opened before filing a PR.

After each multiple-choice grid, we offered a follow-up free-text question for supplementary information. Seven new influential aspects were mentioned multiple times among the 49 free-text responses. Four of them, i.e., project fitness, whether the changes improve the project, the presence of a task description, and code readability, echo the first finding that reviewers appreciate fitness and clarity. The remaining aspects were whether the changes have been discussed in an issue, the atomicity of the changes, and whether a bug is fixed. We found a few specific responses interesting. One respondent stated that they usually do not even review PRs that have a conflict or fail the CI build. Another stated that even a perfect PR would be rejected if it was outside the project's scope. To avoid discouragement, one respondent used more careful language with newcomers. Finally, our last response mentioned that bug fixes that delete more code than they add and pass all tests are merged after a quick review.

Actions when reviewing pull requests

We also asked reviewers how they respond to PRs to increase acceptance. Figure 4.5 shows the responses in terms of how often they use each action. Counting the number of "very often" and "often" votes, the respondents most com-

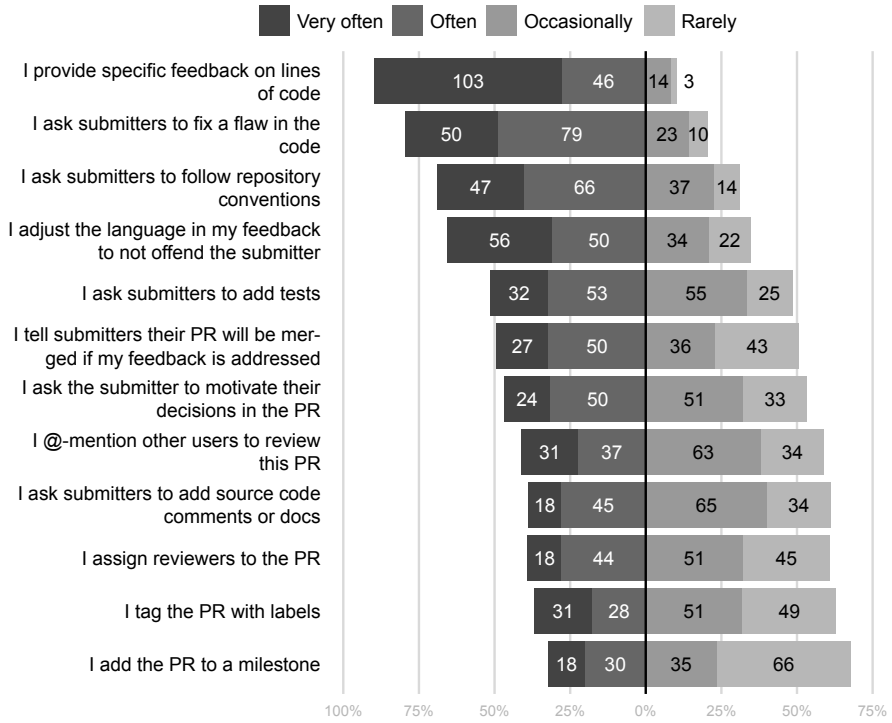

Figure 4.5 – Actions performed by reviewers when reviewing pull requests to increase acceptance, ordered by frequency. Some actions are rephrased for brevity.

monly provide specific feedback on code, ask submitters to fix their code, ask submitters to follow repository conventions, and try not to offend submitters. The remaining actions were considerably less common and used about equally often. Moreover, the most common actions were much more skewed than the least common. This all suggests that the vast majority of reviewers respond in similar ways, specifically, asking submitters to improve their code and follow conventions while trying not to offend them. Other actions are considerably less common but none of them are rare.

We also gave respondents a supplementary free-text question that received 20 responses, 12 of which suggested new actions. None of the actions were mentioned multiple times, reinforcing the idea that reviewer behavior is highly varied. We gain more insights by grouping the actions more coarsely. Four actions accommodated the submitter in some way, e.g., reviewers suggesting code changes that can be automatically applied and merging the PR to fix tiny flaws themselves. This reiterates the finding that reviewers accommodate submitters. Three actions pertained to tests, i.e., testing the PR locally, asking

whether parts not covered by automated tests have been tested, and asking the submitter to run integration tests. The insight is that some reviewers want reassurance that all tests are successful, which is tangentially related to asking submitters to add tests, the fifth most common action.

Key takeaways for RQ4. Reviewers prioritize project fitness in both the first and final review of a PR, and simplicity and clarity are appreciated particularly in the first review. Some aspects are more context-sensitive, e.g., the usage of milestones. Reviewers typically ask for code improvements and more conformance while trying to accommodate submitters. Actions beyond these are less common and more varied.

Chapter 5

Discussion

In this chapter, we discuss the implications of this work (Section 5.1) and put our findings in the context of the related works (Section 5.2). Moreover, we expose the threats to validity (Section 5.3) and provide notes for future surveys on GitHub users (Section 5.4).

5.1 Implications

In this section, we highlight the implications of the results presented in Chapter 4. First, we discuss the efficacy of predicting the acceptance of open pull requests (PRs) with data from closed PRs and give takeaways to developers of PR recommendation tools (Section 5.1.1). Then, we compare the submitter and reviewer responses to our survey and give takeaways for GitHub users looking to increase the acceptance rate of PRs (Section 5.1.2).

5.1.1 Implications for developers of recommendation tools

In Section 4.1.1, we evaluated the prediction of acceptance using classifiers trained on closed GitHub PRs and tested on PRs in three states. The states correspond to when the PR was ready for review, halfway complete, and complete. Then, in Section 4.1.2, we measured the relative importance of each factor in each state. Based on our findings, these are the takeaways for tool developers looking to model the acceptance of closed PRs as a way to recommend open PRs to reviewers.

When evaluating the predictions, we found promising evidence that specifically merges of open PRs can be successfully predicted by training on closed

PRs. Unfortunately, our predictions of non-merges suffered greatly with open PRs, meaning that recommendation tools may want to suppress such predictions or inform their users in such cases of low confidence. This may not be an issue if the penalty for a wrong non-merge prediction is small, which depends on the project the tool is deployed in. On the other hand, we found that developers can reasonably expect exponential prediction gains to all metrics as PRs approach completion.

When measuring factor importance, we found that the top factors for the acceptance of closed PRs were also the most important in the open states. However, sometimes their effect was reversed, i.e., they hurt predictions. This means that models trained on closed PRs can misjudge some factors when predicting open PRs. These factors were the number of approved code reviews, last CI build status, and amount of PR discussion. Developers may want to model these for the occasional benefit or remove them due to the occasional drawback. A more performant but complex solution would be to conditionally weigh factors depending on how near completion the PR appears to be. The good news is that we discovered two factors that contributed a neutral or positive effect across the three states, i.e., the number of events and participants in the PR. This means that some factors can be modeled for additional performance in some cases without any downside.

5.1.2 Implications for submitters and reviewers

In Section 4.2, we presented the results of our survey on GitHub PR users. Specifically, we summarized the main demographic, how submitters prepare PRs and respond to feedback, and how reviewers judge and respond to PRs. In this section, we connect and contrast the submitter and reviewer responses, and conclude with takeaways for users who wish to increase the acceptance of the PRs they develop or review.

We discovered three major patterns that both parties focused on from submission to closing. The first pattern was conformance. Making sure to follow the submission guidelines and coding conventions was among the top four preparation methods for submitters. Despite their emphasis, the third most common reviewer action was to ask for more conformance, and consequently, the fourth most common submitter response was to increase conformance. The second pattern was clarity and intent. Submitters gave a clear PR title and description most often and they explained the value of their PR moderately often. Reviewers still had to ask for motivation moderately often and the third most common submitter response was to clarify or elaborate on their intent. The

third pattern was tests and source code comments or documentation. Like with the other major patterns, these aspects were similarly prevalent in the preparation, review, and feedback response.

We found three less common and looser patterns. First, milestones and labels were barely used by both parties, probably because milestones are rare and bots can apply labels. Second, submitters were less careful with offending the reviewer compared to the other way around. However, reviewers considered the submitter's attitude to be relatively less important in the final decision. Third, submitters commonly tagged their PRs with an issue, and reviewers perceived this as relatively less important.

As previously mentioned, Section 4.2 shows the common behavior and values of submitters and reviewers. Because most of our responses came from active PR users, the findings are particularly useful for GitHub novices looking to learn how to contribute to open-source projects. The findings may also serve as a good foundation for a set of submission guideline items. In this section, we discovered commonalities and discrepancies between the submitters and reviewers. The major themes of interest were conformance, clarity and intent, and tests and source code comments or documentation. Despite being valued by both parties, reviewers tended to ask for improvements in these aspects. The implications are that submitters should consider emphasizing these aspects further and reviewers may want to update their guidelines accordingly.

5.2 Comparison to related work

In this section, we explore how our results from Chapter 4 compare to the related works. We examine how our PR acceptance models fared against prior models in Section 5.2.1. Moreover, we compare the importance of our PR factors to prior findings in Section 5.2.2. Then, in Section 5.2.3 and Section 5.2.4, we contrast our survey results on submitter and reviewer behavior and values against previous surveys. Table 3.1 and Table 3.2 contain a cursory summary of the works mentioned in this section.

5.2.1 Predictions on pull request acceptance

Below, we compare how our models fared against the related works that also modeled PR acceptance. Some works did not specify which class the class-specific metrics pertained to, and we assume the positive class (merged) in such cases. Our work is unique in the sense that we evaluated our models when predicting on PRs in three states, as opposed to testing on closed PRs

exclusively. Therefore, we focus only on our results pertaining to closed PRs to make the comparisons fair.

Gousios et al. [4] modeled acceptance and their choice of algorithms and performance metrics are also featured in this work. First, we compare the Logistic Regression and Naive Bayes models. We achieved much higher accuracy, slightly lower merge precision, and much higher merge recall. Relative to their Random Forest model, we achieved a similar merge precision and slightly better accuracy and merge recall. Our three models scored much lower AUC. This comparison comes with the caveat that their data is the oldest by far in this section, including PRs from 2012.

Ortu et al. [18] evaluated different sets of factors using two algorithms and several metrics. We chose to focus on the models that achieved the highest AUC scores for the comparisons. Overall, our Logistic Regression and Random Forest models achieved higher merge scores, lower non-merge, and lower AUC scores, to varying extents. Again, we see that our merge-specific scores are superior while our AUC score is inferior.

The remaining works all featured only one algorithm in their prediction results. Zampetti et al. [17] created two Random Forest models and we achieved better precision and recall in both classes, better MCC, and similar AUC. Dey and Mockus [16] used the same algorithm and our corresponding model reached a better merge recall but worse non-merge recall and AUC. Finally, Yu et al. [5] achieved a far superior AUC score compared to our Logistic Regression model.

In summary, our models tended to be more accurate and successful in predicting merges, with the opposite result for non-merges. Our models were also less well-rounded, as indicated by our consistently beaten AUC scores. Zampetti et al. [17] was the only work we decidedly outperformed. These findings make sense considering our skewed class distribution (see Section 5.3) and deliberate exclusion of important PR factors (see Section 3.2.3).

5.2.2 Feature importance

Numerous studies have measured the importance of their features when modeling PR acceptance. Below, we connect and contrast our findings on this topic to the related works. Recall that we measured importance when predicting on PRs in three states, unlike the related works that tested on closed PRs exclusively. To ensure fair comparisons, we focus only on our results pertaining to closed PRs.

We shared many features with two particular works, the first being Zampetti et al. [17]. They found their four CI features important to varying degrees, corroborating our results. We also agree on the relative unimportance of the number of touched files and test inclusion. A vague connection can be made between their third most important feature, i.e., PR age, and our fourth, i.e., the number of timeline events. We diverge on the amount of discussion as they found it negligible and it was our third ranking feature. Whether the PR aims at fixing an issue was very important in their analysis, while our corresponding feature was negligible. Lastly, we found the number of commits slightly more valuable. Yu et al. [5] also included many similar features. They found CI important, the number of commits moderately important, and @-mentions and fixes unimportant, all of which corroborate our findings. In terms of disagreement, we found the amount of discussion relatively more important and tests relatively less so.

Two works measured one dominant feature while the rest were far less influential. Because this makes convincing comparisons more difficult, we only point out the major similarities and differences. For Gousios et al. [4], the extent to which the PR modifies recently modified code was dominant. Our results agreed that conflict was negligible but disagreed on the importance of the amount of discussion, a top feature for us and a bottom feature in their work. For Dey and Mockus [16], the age of the PR was dominant. Again, this is vaguely related to one of our top features, i.e., the number of timeline events, albeit we did not find it nearly as influential. Our strongest agreement was that the number of fixes is negligible.

The amount of discussion was a commonly shared feature. Beyond the aforementioned works, Iyer et al. [11], Tsay et al. [14], and Kononenko et al. [6] modeled this feature and obtained different results. The former two works found discussion relatively unimportant, contrary to our finding, while the latter found it important, which we also did.

In summary, we observed mixed agreement with the related works. Our strongest corroboration was with two previous findings that some CI features are important. Our strongest contradiction was that we found the amount of discussion substantially more important than several other works. The remaining comparisons were vague, only corroborated once, or conflicted across the related works. These results may be explained in part by our unconventional method of measuring importance, i.e., the drop-column strategy (described and motivated in Section 3.2.5).

5.2.3 Submitter actions

Part of our survey on GitHub users inquired submitters about their methods when preparing PRs and responding to feedback. Gousios et al. [10] is the only survey that is directly related, and it inquired about actions before and after coding a PR. Overall, we agreed sharply with their results. Their two most common actions after coding their PR were running tests and formatting the PR according to the project guidelines. Our corresponding actions, i.e., passing the CI build, testing locally, and following guidelines, were also common in our responses. Checking for duplicate work was moderately common collectively across their two questions, which was also the case for us. We found arguably just one disagreement. Before coding their PR, respondents looked up the guidelines occasionally, while our related action ranked second. However, this makes sense considering our demographic differences. They invited the top contributors of each selected project, who are naturally familiar with the guidelines, while we invited submitters of any experience level.

5.2.4 Reviewer actions and opinions

Part of our survey on GitHub users asked reviewers how they respond to PRs and judge them in the first review and final decision. In this section, we compare our results to the findings of related surveys.

The most closely related work is a large-scale survey by Gousios et al. [2] that invited top integrators, i.e., users who make merge decisions. First, the respondents shared how they review PRs, to which they also found that inline comments attached to code are common. Then, they shared aspects that affect the merge decision. Code quality ranked first which arguably encapsulates several aspects we asked about in our survey, i.e., code complexity, test inclusion, and source code comments or documentation. Our results ranked these at least moderately high. Their other top aspects were code style, project fit, and technical fit, all of which are related to project fitness which was the top concern in our results. A few aspects could be compared more directly. Tests, documentation, discussion, and the submitter's track record all achieved similar influence in our surveys. Some of their results appear to contradict themselves, e.g., correctness was a bottom aspect despite fitness being a top feature, but this was likely because respondents could only choose three aspects and thus opted for the broader ones. We point this out as it explains the contradicting results. The only contradiction we could not explain this way was to the responsiveness, which scored relatively lower in the related work. Finally, the integrators were asked how they judge PR quality. Our results corroborated

with theirs strongly, e.g., we found conformance and tests important and documentation and size moderately important. One disagreement was that the test result was their fifth most influential aspect, while our corresponding aspect, i.e., the CI status, ranked first.

In a case study, Kononenko et al. [6] asked developers what aspects influenced the merge decision. They found PR size moderately to highly important overall, and our results agree to some extent. We strongly agree with their findings that the length of the discussion and the submitter's experience and affiliation are relatively unimportant. Respondents also shared the signals of PR quality in an open question. Although their results are difficult to compare directly, we note that they also found project fitness and clarity valuable. Their survey also inquired how developers reviewed PRs, however, their findings cannot be meaningfully connected to ours.

We have two final related works that could only be compared in a few ways. Steinmacher et al. [3] asked integrators to share the reasons for rejecting PRs. A very common reason was not following repository guidelines, and our results agree strongly, considering that compliance was our second most influential aspect in the final decision. When Zampetti et al. [17] asked 13 CI practitioners whether the last CI build status affects the merge decision, the responses were a resounding *yes*. We agree sharply considering that our corresponding aspect ranked first.

Based on the comparisons above, we notice a relatively strong consensus with the related works. Ordered by overall influence, the aspects that corroborated with multiple sources were project fitness, CI, tests, PR size, discussion, and submitter experience.

5.3 Threats to validity

There are two threats to validity that apply to both our quantitative and qualitative analyses. First, all of the results are constrained by the particular PR data and survey responses. That said, we employed strategies to make the results more widely applicable, e.g., restricting the number of PRs per repository and inviting users from many different repositories. Second, because PRs were collected from 2019 at the earliest and the survey was conducted in 2022, both data sets may be influenced by the impact of COVID-19.

A clear threat to the validity of our PR data is errors in GitHub's API and our implementations. Crucially, some PRs may appear non-merged even

though their commits were in fact merged[1]. This has been acknowledged in the past and a strategy has been proposed to catch these false negatives [4], however, we did not use it to manage the scope of this work. In addition, several related works did not use it [17, 27, 5]. We manually inspected some PRs to find out what caused the large factor values in Table 3.7 and found that the main cause was bot activity, e.g., posting many duplicate comments[2]. Moreover, two factors (*Changed_Files* and *Test_Files*) were overestimated because we failed to discount some commits generated by automatic processes. The number of PR commits was capped at 250 by GitHub's API and the number of @-mentions was inflated due to what appeared to be duplicate events. See Section 1.4 for the full report on this analysis.

Another threat to our PR data is the skewed class distribution. Our rate of merged PRs was 85%. For reference, past works have reported merge rates between 53% [12] and 85% [4], however, the former work was a case study and the latter used the previously mentioned strategy to detect additional merges. Regardless, we can see that the merge rate depends on the data selection and ours may have been biased in favor of merged PRs. One study [4] used a merge rate threshold to exclude repositories with unusually few merges but we did not use it to ensure broad applicability of our findings and because several related works did not use it [15, 5, 11].

The final threat to the quantitative part of our work is the modeling. Ideally, we would have trained our models on past PRs and tested them on future PRs. However, we only divided our data using random sampling, meaning that we occasionally predicted on past PRs based on future PRs. This is known as *data leakage* and means that our prediction results do not necessarily reflect how the models would perform in a real use case. One study successfully addressed this threat [16] but it is outnumbered by ones that failed to do so [6, 18, 17], including ours.

Concerning our survey, the questions and multiple-choice options may have been phrased ambiguously, in turn confusing respondents and hurting the legitimacy of their answers. In fact, one respondent pointed out how the presence of conflict could be interpreted as code conflict or conflict of interest. Furthermore, we limited the multiple-choice options to keep the questions from being unreasonably long but at the expense of potentially losing insights. Finally, we may have filtered responses inadequately, leaving undesirable responses in the data set.

[1]Example: `https://github.com/curl/curl/pull/8219`. Last access 2022-11-30.
[2]Example: `https://github.com/openshift/origin-aggregated-logging/pull/2053`. Last access 2022-11-30.

5.4 Notes for future GitHub surveys

In this section, we share notes for researchers interested in executing their own survey on GitHub users in future studies. Note that the complete survey, the template for generating invitations, and the raw and filtered responses are available in Section 1.4.

First off, we encourage future surveys to have a "not applicable" multiple-choice option. Some of our respondents pointed out this missing option as they were not responsible or lacked permission for some actions, e.g., bots may be responsible for labeling and some users may be prohibited from assigning reviewers. We also encourage the inclusion of a demographic question asking whether the respondent was invited directly or by a peer, assuming the survey is open. Two related works [2, 10] included this and both found that a significant amount of responses came from third-party advertising. Corroborating or contrasting their finding would help future creators estimate the value of having an open survey.

The participation rates of our survey provide some lessons. Out of 2,000 invited GitHub users, 3.9% (78) manually replied to the email, 2.3% (45) requested to see the results of the study, and 0.2% (3) shared the survey. Out of 68 invited authors of the related works, 8.8% (6) manually replied, 2.9% (2) requested the results of the study, and 2.9% (2) shared. The final participation rate was 9.9% $(204/2068)$. Based on these rates, we learn that acceptable participation rates can be achieved despite not filtering no-reply emails. Also, the authors participated more which was our expectation.

We would like to highlight specific replies to our survey invitations. Two replies complained about the length of our survey, one of which quit before completion. Unfortunately, we could not retrieve the completion time of the responses, but for context, we had 17 questions (12 optional, 5 mandatory) and the longest question had 21 multiple-choice options. Five replies refrained from answering due to privacy concerns with Google, doubting the legitimacy of the work, or not wanting to click a random link. Two replies explained that they do not code, e.g., one worked on documentation, which we did not consider. Finally, offering to share our survey results may have increased participation considering the appreciation expressed in the replies.

Chapter 6

Conclusion

In this work, we investigated the temporal side of pull request (PR) acceptance on GitHub. PR acceptance has been modeled in previous works. However, there has been no comparison between predicting on open and closed PRs. This comparison could be useful to developers of tools that recommend PRs as it could help them gauge the efficacy of predicting on open PRs as they near completion, which is the real use case. We executed a quantitative study in which we tested and evaluated our models on PRs in three states: when they were ready for review, halfway complete, and complete. Moreover, we measure the importance of each factor in each state, which may help tool developers improve their underlying models.

Several surveys have been conducted on the users of PRs on GitHub, however, some elements of their usage remain unknown. Additional insights could help users increase the acceptance of the PRs they develop and review. We executed a qualitative study that consisted of an open survey on submitters and reviewers of PRs which covered their usage from the preparation phase to its merge decision. Specifically, we asked submitters how they prepare PRs and respond to feedback, and reviewers how they respond to PRs and judge them in the first and final review.

Implications for developers of pull request recommendation tools. Our modeling results show that tools trained on closed PRs can suffer significantly when making predictions on open PRs compared to closed PRs. While merges of PRs early in their lifetime can be predicted successfully, non-merge predictions can occasionally be worse than naive models. The opposite may be the case if the merge rate of the target repository is low. However, one can expect exponential gains to predictions as PRs near their completed state. Moreover, tools trained on closed PRs can wrongly estimate the value of specific factors in open PRs. Our top factors for the acceptance of closed PRs were also the

most important in the open states, but sometimes their effect was reversed, i.e., they hurt predictions. These factors were the number of approved code reviews, last CI build status, and amount of discussion. Developers may wish to model these for the occasional benefit, ignore them to avoid the occasional drawback, or conditionally weigh them for an improved but more complex solution. Finally, we discovered two factors, i.e., the number of events and participants on the PR, that could be modeled with no downside and an occasional performance boost.

Implications for users of pull requests. Based on our survey results, novices can learn how to improve their chances of having their PRs accepted. Submitters and reviewers agreed that PRs should have clear intent and fit the project. This includes a clear title and description, tests, source code comments or documentation, a passing CI build, and conformance to submission guidelines and coding conventions. Experienced users looking to refine their development or code review process may also appreciate these pointers. This applies especially to users who are active on a small number of repositories and want more general advice to diversify their activity. Additionally, these findings could serve as a solid foundation for a set of submission guideline items. The final takeaways come from the common responses to PRs. Despite the efforts by submitters, reviewers very often ask for code improvements and more conformance, moderately often ask for tests, and occasionally ask for source code comments or documentation. Experienced submitters may want to emphasize these aspects further and writers of submission guidelines should consider updating their guidelines accordingly.

Secondary contributions. Planning and executing our study created several secondary contributions. First, we summarized the PR factors that had been previously modeled. Then, we devised a novel method for collecting temporal PR data, which was implemented and used to collect temporal data from 100,000 PRs. Finally, we designed a survey on users of PRs on GitHub which received 204 responses. Section 1.4 lists our contributions, all of which are available publicly or in this thesis.

Future work. There are multiple avenues for future work. A possible conceptual replication of our modeling is to collect open PRs and their merge status when they have closed. This removes the need to recreate open PRs based on their closed counterpart. One can study PRs that initially look hopeless but eventually succeed and vice versa. Our PR data can be used to compare how the characteristics of PRs change across their lifetime, or perhaps compare short-lived to long-lasting PRs. Finally, our survey responses can be grouped by demographic to contrast different kinds of users.

References

[1] G. Zhao, D. A. da Costa, and Y. Zou, "Improving the pull requests review process using learning-to-rank algorithms," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2140–2170, 2019.

[2] G. Gousios, A. Zaidman, M.-A. Storey, and A. Van Deursen, "Work practices and challenges in pull-based development: The integrator's perspective," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 358–368.

[3] I. Steinmacher, G. Pinto, I. S. Wiese, and M. A. Gerosa, "Almost there: A study on quasi-contributors in open source software projects," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 256–266.

[4] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 345–355.

[5] Y. Yu, G. Yin, T. Wang, C. Yang, and H. Wang, "Determinants of pull-based development in the context of Continuous Integration," *Science China Information Sciences*, vol. 59, no. 8, pp. 1–14, 2016.

[6] O. Kononenko, T. Rose, O. Baysal, M. Godfrey, D. Theisen, and B. De Water, "Studying pull request merges: A case study of Shopify's Active Merchant," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, 2018, pp. 124–133.

[7] W. Zou, J. Xuan, X. Xie, Z. Chen, and B. Xu, "How does code style inconsistency affect pull request integration? An exploratory study on 117 GitHub projects," *Empirical Software Engineering*, vol. 24, no. 6, pp. 3871–3903, 2019.

[8] E. Van Der Veen, G. Gousios, and A. Zaidman, "Automatically prioritizing pull requests," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 357–361.

[9] C. Maddila, S. S. Upadrasta, C. Bansal, N. Nagappan, G. Gousios, and A. van Deursen, "Nudge: Accelerating overdue pull requests towards completion," *arXiv preprint arXiv:2011.12468*, 2020.

[10] G. Gousios, M.-A. Storey, and A. Bacchelli, "Work practices and challenges in pull-based development: The contributor's perspective," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 285–296.

[11] R. N. Iyer, S. A. Yun, M. Nagappan, and J. Hoey, "Effects of personality traits on pull request acceptance," *IEEE Transactions on Software Engineering*, 2019.

[12] V. Lenarduzzi, V. Nikkola, N. Saarimäki, and D. Taibi, "Does code quality affect pull request acceptance? An empirical study," *Journal of Systems and Software*, vol. 171, p. 110806, 2021.

[13] M. C. O. Silva, M. T. Valente, and R. Terra, "Does technical debt lead to the rejection of pull requests?" *arXiv preprint arXiv:1604.01450*, 2016.

[14] J. Tsay, L. Dabbish, and J. Herbsleb, "Influence of social and technical factors for evaluating contribution in GitHub," in *Proceedings of the 36th international conference on Software engineering*, 2014, pp. 356–366.

[15] T. Dey and A. Mockus, "Which pull requests get accepted and why? A study of popular NPM packages," *arXiv preprint arXiv:2003.01153*, 2020.

[16] ——, "Effect of technical and social factors on pull request quality for the NPM ecosystem," in *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020, pp. 1–11.

[17] F. Zampetti, G. Bavota, G. Canfora, and M. Di Penta, "A study on the interplay between pull request review and Continuous Integration builds," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 38–48.

[18] M. Ortu, G. Destefanis, D. Graziotin, M. Marchesi, and R. Tonelli, "How do you propose your code changes? Empirical analysis of affect metrics of pull requests on GitHub," *IEEE Access*, vol. 8, pp. 110 897–110 907, 2020.

[19] D. M. Soares, M. L. de Lima Júnior, L. Murta, and A. Plastino, "Rejection factors of pull requests filed by core team developers in software projects with high acceptance rates," in *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2015, pp. 960–965.

[20] Y. Guo and P. Leitner, "Studying the impact of CI on pull request delivery time in open source projects—a conceptual replication," *PeerJ Computer Science*, vol. 5, p. e245, 2019.

[21] J. H. Bernardo, D. A. da Costa, and U. Kulesza, "Studying the impact of adopting Continuous Integration on the delivery time of pull requests," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 2018, pp. 131–141.

[22] Y. Yu, H. Wang, V. Filkov, P. Devanbu, and B. Vasilescu, "Wait for it: Determinants of pull request evaluation latency on GitHub," in *2015 IEEE/ACM 12th working conference on mining software repositories*. IEEE, 2015, pp. 367–371.

[23] M. L. de Lima Júnior, D. Soares, A. Plastino, and L. Murta, "Predicting the lifetime of pull requests in open-source projects," *Journal of Software: Evolution and Process*, p. e2337, 2021.

[24] D. Moreira Soares, M. L. de Lima Júnior, L. Murta, and A. Plastino, "What factors influence the lifetime of pull requests?" *Software: Practice and Experience*, vol. 51, no. 6, pp. 1173–1193, 2021.

[25] M. I. Azeem, Q. Peng, and Q. Wang, "Pull request prioritization algorithm based on acceptance and response probability," in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2020, pp. 231–242.

[26] C. Maddila, C. Bansal, and N. Nagappan, "Predicting pull request completion time: A case study on large scale cloud services," in *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 874–882.

[27] M. I. Azeem, S. Panichella, A. Di Sorbo, A. Serebrenik, and Q. Wang, "Action-based recommendation in pull-request development," in *Proceedings of the International Conference on Software and System Processes*, 2020, pp. 115–124.

[28] T. Parr, K. Turgutlu, C. Csiszar, and J. Howard, "Beware default Random Forest importances," 2018. [Online]. Available: https://explained.ai/rf-importance/index.html

TRITA-EECS-EX-2023:15