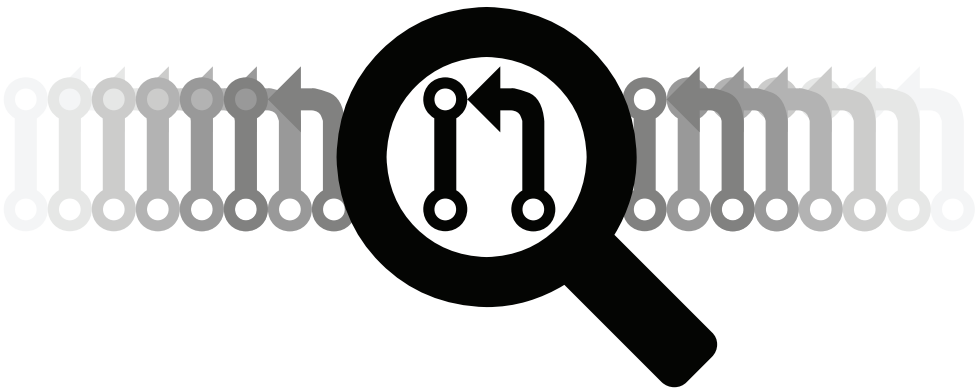
Prioritizing pull requests

Version of June 17, 2015



Erik van der Veen

Prioritizing pull requests

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Erik van der Veen
born in Voorburg, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Q42
Waldorpstraat 17F
2521 CA
The Hague, the Netherlands
www.q42.com

Cover picture: Finding the pull request that needs the most attention.

Prioritizing pull requests

Author: Erik van der Veen
Student id: 1509381
Email: erikvdv1@gmail.com

Abstract

Previous work showed that in the pull-based development model integrators face challenges with regard to prioritizing work in the face of multiple concurrent pull requests.

We identified the manual prioritization heuristics applied by integrators and extracted features from these heuristics. The features are used to train a machine learning model, which is capable of predicting a pull request's importance. The importance is then used to create a prioritized order of the pull requests. Our main contribution is the design and initial implementation of a prototype service, called PRioritizer, which automatically prioritizes pull requests. The service works like a priority inbox for pull requests, recommending the top pull requests the project owner should focus on. It keeps the pull request list up-to-date when pull requests are merged or closed.

In addition, the service provides functionality that GitHub is currently lacking. We implemented pairwise pull request conflict detection and several new filter and sorting options e.g. the pull request's size.

A preliminary user study showed that the PRioritizer service, although it is positively evaluated as a whole, needs to give users more insight into how the priority ranking is established to make it really useful.

Thesis Committee:

Chair:	Prof. dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. A.E. Zaidman, Faculty EEMCS, TU Delft
Company supervisor:	Ing. G. Goossens, Q42
Committee Member:	Dr. G. Gousios, Faculty of Science, Radboud University

Preface

This thesis is a product of my graduation project for the Master of Science degree at the Delft University of Technology. I would like to thank my supervisors at the university, Georgios Gousios and Andy Zaidman, for their terrific guidance, support and understanding.

Of course I would like to thank Q42 for the great opportunity to conduct my research in a highly active development environment. Especially, I want to thank my company supervisor, Guus Goossens, for sharing his experiences and tips with me. Also, I want to thank all other colleagues for their interesting questions and conversations.

I would like to thank Audris Mockus for his influence in the design of the prioritization algorithm.

Finally, I want to thank my wonderful girlfriend Heidi and my parents for their unconditional love, support and patience.

Erik van der Veen
Delft, the Netherlands
June 17, 2015

Contents

Preface vii

Contents ix

List of Figures xi

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Objective . . . 2
- 1.3 Research Questions . . . 3
- 1.4 Research Approach . . . 3
- 1.5 Research Context . . . 4
- 1.6 Outline . . . 4

2 Background and Related Work 5
- 2.1 Distributed version control systems . . . 5
- 2.2 Pull requests . . . 6
- 2.3 GitHub . . . 7
- 2.4 GHTorrent . . . 8
- 2.5 Machine Learning . . . 8
- 2.6 Bug triage . . . 9
- 2.7 Pull request usage . . . 10
- 2.8 Pull request triage . . . 11

3 Analysis 13
- 3.1 Survey participants . . . 13
- 3.2 Questions . . . 13
- 3.3 Processing . . . 14
- 3.4 Results . . . 14
- 3.5 Discussion . . . 18

4 Modeling **21**
4.1 Features . . . 21
4.2 Training data . . . 24

5 Implementation **27**
5.1 Architecture . . . 28
5.2 Programming languages . . . 28
5.3 Watcher . . . 28
5.4 Analyzer . . . 29
5.5 Predictor . . . 30
5.6 Visualizer . . . 31
5.7 Extensibility . . . 33
5.8 Usage . . . 33

6 Experimental Setup **37**
6.1 Machine learning tuning . . . 37
6.2 Evaluation participants . . . 38

7 Performance and Evaluation **39**
7.1 Algorithm performance . . . 39
7.2 Application performance . . . 41
7.3 Evaluation . . . 42

8 Discussion and Future Work **47**
8.1 Results . . . 47
8.2 Threats to validity . . . 47
8.3 Future work . . . 49

9 Conclusion **51**
9.1 Conclusions . . . 51
9.2 Contributions . . . 52

Bibliography **53**

A Glossary **57**

B Service log **61**

C Project list **63**

D Queries **67**

E Evaluation survey **69**

List of Figures

2.1 Git versus SVN 6
2.2 Usage of VCSs in 2014 6

3.1 Work field distribution 15
3.2 Classified prioritization heuristics 15
3.3 Heuristic subcategories. 17
3.4 Heuristic cross correlation 19
3.5 Work field correlation 19
3.6 Newly created pull request correlation 19
3.7 Pull request triage method distribution 20

4.1 Branch diagrams with merge conflicts. 24
4.2 Time windows 25

5.1 Diagram of the architecture 27
5.2 The user interface 32
5.3 The decorator pattern 34

7.1 Plot of feature importance 41
7.2 User evaluation of individual feature usefulness 43
7.3 User evaluation of the service usefulness 43
7.4 User evaluation of future features usefulness 44
7.5 User evaluation of the service aspects 44

Chapter 1

Introduction

In software engineering, software repositories are often managed by version control systems (VCSs). VCSs help developers to implement and maintain large systems by letting them collaborate and work on the same project at the same time. Team members are able to work freely on any file, the VCS will later allow developers to merge all the changes into a common version. VCSs also keep track of previous versions of each file, it enables developers to fairly easy restore a file to its previous state if a bug was introduced in a newer version. As a side-effect, it helps developers to understand how the project evolved between versions.

During the last several years, *distributed* version control systems (DVCSs), such as Git, emerged in the software field [20]. These systems have technical features that allow contributors to work in new ways.

Traditionally, *centralized* version control system (CVCS), like Subversion (SVN) were frequently used in open source projects. One of the main disadvantages of CVCSs is that contributors must have writing permissions in order to perform basic tasks, such as, creating a branch or publishing changes with full revision history [1]. This limitation affects participation and authorship for new contributors. Due to its characteristics, a DVCS is not limited by these issues and non-core developers can easily contribute to a project [25].

The use of a DVCS like Git, enables a workflow method involving *pull requests*. Pull requests as a distributed development model in general, and as implemented by GitHub in particular, form a new method for collaborating on distributed software development. The novelty lays in the decoupling of the development effort from the decision to incorporate the results of the development in the code base. By separating the concerns of building artifacts and integrating changes, work is cleanly distributed between a contributor team that submits changes to be considered for merging and a core team that oversees the merge process, provides feedback, conducts tests, requests changes, and finally accepts the contributions [9].

Recently, DVCSs were widely adopted, especially by the open source community. Many well-known open source projects migrated from CVCSs to DVCSs [27]. Along with the appearance of DVCSs also new project hosting and collaborating platforms built around DVCSs emerged. One of the most known and popular platforms is GitHub, which is built around Git.

GitHub facilitates a simple and clear way of creating, reviewing and discussing pull requests [17]. It enables and encourages even the less skilled developers to view the project's

source code, fork it, send in a pull request and enter the discussion.

1.1 Motivation

While working with DVCSs and pull requests provide several advantages for non-core developers e.g. an easy method to contribute to an open source project, it can also cause some difficulties for the core developers. Because of the low contribution barrier, many projects on GitHub have dozens of open pull requests and receive new ones every day. In some cases, the number of open pull requests can even go up to several hundreds[1]. Not only for those large projects, but also for projects with a smaller amount of pull requests, it quickly becomes quite a challenge to decide which ones are valuable for the project and deserve the attention of the core developers the most.

GitHub contains a wide variety of different projects. They do not only differ in purpose and content, but also in usage. The majority of the projects on GitHub has only one developer, often with no pull requests at all [9], so they might not benefit from a prioritization of their pull requests. However, there are still a lot of projects that have, at any point in time, dozens of open pull requests.

A survey among 749 developers was conducted by Gousios et al. [12] to investigate the work practices and challenges in a pull-based development environment of high volume projects. The majority of the respondents (82%) use a form of manual prioritization of incoming pull requests. Integrators, the developers who process the contributions, often apply multiple criteria to prioritize contributions e.g. the pull request's age, size and the contributor's track record. The study also shows that for many of the respondents one of the key challenges is finding free time to devote on handling pull requests as managing contributions is not their main job.

Most collaborating platforms, such as GitHub, only provide a simple way to order and filter pull requests e.g. on age, author name, number of comments, etc. With the help of a tool that provide the core developers with an automatically and intelligently prioritized list of pull requests, they could save a significant amount of time. Especially if there are a lot of open pull requests. To the best of our knowledge, such a tool does not exist yet.

1.2 Objective

In this thesis, we present the design, initial implementation and evaluation of a prototype pull request prioritization tool, the PRioritizer. The PRioritizer tool works as a priority inbox [5] for pull requests: it examines all open pull requests and presents project integrators the top pull requests that potentially need their immediate attention. The ranking is determined by a machine learning algorithm which is trained with historical project data. The tool also offers an alternative view to GitHub's pull request interface, which allows developers to sort open pull requests on a multitude of criteria, ranging from the pull request's age to its number of conflicts with other open pull requests (pairwise conflicts).

[1] At the time of writing (April 22th, 2015) the AngularJS project had 310 open pull requests on GitHub.

The PRioritizer is a service-oriented architecture build on top of the GHTorrent project [8]: it uses GHTorrent's data collection mechanisms to react in near real-time to changes in pull request state.

The study upfront of the developing of the PRioritizer tool leads us to the research questions discussed in the next section.

1.3 Research Questions

To account for the different uses of the pull-based model [9], we use machine learning models at the core of the implementation. Each specific project has its own trained model, based on its history, to rank the pull requests. To train a machine learning model which pull requests are more important than others, it needs to know what the quantifiable properties of a pull request are i.e. we have to extract features from the pull requests. This leads to the following question:

RQ1 What features can be used for prioritizing pull requests?

Once we know how to classify pull requests we can think about how we design a pull request prioritization system. In order to create a useful system that prioritizes pull requests it must be capable of prioritizing many pull requests for many projects. To achieve this, we have to answer the next question:

RQ2 How to implement a scalable system which prioritizes pull requests?

When we know the answers to the previous questions we can attempt to implement a tool that is capable of automatically prioritize pull requests. Achieving a good prioritization of the pull requests is the main objective of our research. In order to solve this problem we need to investigate our main research question:

RQ3 Is it possible to obtain a useful automatic prioritization of pull requests?

1.4 Research Approach

To answer our research questions we use different research techniques. Before we answer RQ1, we perform an statistical analysis on the results of a large-scale qualitative study to manual prioritization techniques. From the results we extract features which are suitable for prioritization.

With a prove-by-design we answer RQ2. We actually implemented a scalable application, the PRioritizer service, that automatically prioritizes pull requests. The service fetches pull request data from quantitative data sources.

Finally, we perform an analysis on the results of a small qualitative study. We asked integrators to evaluate the PRioritizer service on their own repositories. With their feedback we give an answer to our main question RQ3.

1.5 Research Context

Both the research phase and the implementation phase were conducted at Q42[2] in The Hague. Q42 is a company that creates software for the web, mobile devices and the Internet of Things. More than 50 programmers, who use GitHub on a daily basis for their project hosting and collaboration, are working at Q42. They develop both open and closed source software and offer a chance to be right in the center of an active software engineering environment.

> *"Some people call us 'a tech-savvy internet agency'. Others just call us nerds. We call ourselves nerds too, to be honest. We just happen to really love good programming."*
>
> - Q42

1.6 Outline

First of all, we give some background information about the used technologies and related work in chapter 2. In chapter 3 we present the results of a user survey about manual prioritization heuristics. How we model the pull request data by extracting features from the heuristics, so that it is suitable for prioritization is described in chapter 4. Chapter 5 discusses the implementation of the PRioritizer service. In chapter 6 our experimental setup is described. The performance and an evaluation of our service in this setup is presented in chapter 7. We conclude with a discussion in chapter 8. And finally, we summarize our findings in chapter 9.

[2] q42.com

Chapter 2

Background and Related Work

To get a proper understanding of the topics discussed in this thesis we provide background information on several concepts and techniques in this chapter. Along with the background information, we also discuss related work that touches some of the topics that help to get to know the challenges ahead.

2.1 Distributed version control systems

When a project uses a distributed version control system (DVCSs) to version its source code, every developer has a local copy of the entire history, hence a distributed system. A consequence is that a developer can make changes and commit those to his local repository without contacting any server. While the system itself is distributed, most projects also use a central server (e.g. on GitHub) that stores a mirror of the repository.

Once the developer is satisfied with his set of changes he can push them to the server. Commits that are pushed by other developers in the meantime have to be pulled by the developer before he can push his own commits to the server. However, just like with CVCS a developer that wants to push changes to the server must have writing permissions on the server's repository. When contributing to open source projects this is not desirable, as complete strangers with writing permissions can also do a lot of harm to the project. DVCSs aim to improve this issue by introducing a new workflow involving pull requests.

Examples of popular DVCSs are Git[1] and Mercurial[2]. As Rodriguez-Bustos et al. [27] describe, many projects are migrating from CVCSs, like SVN to DVCSs like Git (see chapter 1 for a brief introduction about CVCSs). Since 2014, Git has finally surpassed SVN to be the top code management tool used by software developers, see table 2.1 and figure 2.1. A third of the developers (33.3%) say that they use Git as their primary code management tool compared to 30.7% using Subversion, see figure 2.2. In the remainder of this thesis we concentrate on Git and GitHub, as they are currently the most used (distributed) version control system and platform respectively.

[1] git-scm.com
[2] mercurial.selenic.com

Year	BitBucket	CVS	Git	GitHub	Mercurial	SVN	None	Other
2009		20.0%	2.4%			57.5%	4.7%	15.4%
2010		12.6%	6.8%		3.0%	58.2%	5.6%	13.7%
2011		13.3%	12.8%		4.6%	51.3%	4.3%	13.7%
2012	0.5%	8.9%	23.2%	4.4%	4.3%	46.0%	3.8%	10.6%
2013	1.5%	4.5%	30.3%	6.0%	3.8%	37.8%	5.2%	11.1%
2014	3.9%	3.7%	33.3%	9.6%	2.1%	30.7%	6.4%	10.3%

Table 2.1: Usage of VCSs. The use of different version control systems by developers over the years [29, 30, 31, 32, 33, 34].

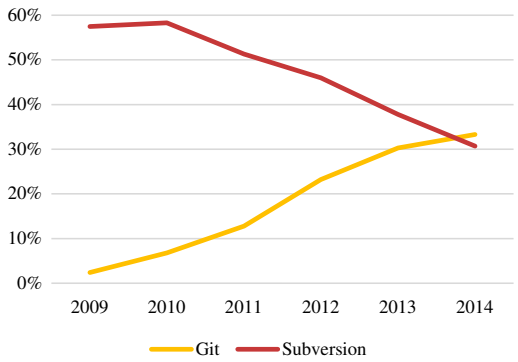


Figure 2.1: Git versus SVN. The use of Git and SVN over the years [34].

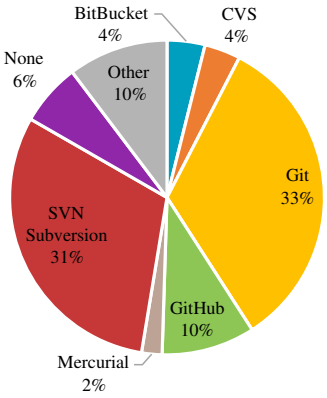


Figure 2.2: Usage of VCSs in 2014. The use of different version control systems by developers in 2014 [34].

2.2 Pull requests

Pull requests solve the issue of contributing to a project without writing permissions to the project's main repository. The development effort is decoupled from the decision to incorporate the results of the development in the code base.

A contributor can, just like a core developer, clone a public project, resulting in a local fork of the project. Changes can be made and pushed to his local fork. All of these actions only require reading permissions on the server's repository. When the contributor is satisfied with his changes he *requests* that his changes should be *pulled* back into the main repository. Thus, a pull request is simply a message to a core developer containing the address of the contributor's repository with the new changes. If such a *pull request* is received by a core developer, it will be reviewed and tested. Potential modifications, e.g. adding more test code or fixing bugs, will be discussed. Finally, when there are no problems, the changes are approved and the pull request is pulled and merged into the project by one of the core developers. Pull requests reduce the amount of friction for new contributors and allow people to work independently without upfront coordination, which is ideal for open source projects.

2.3 GitHub

Probably one of the most known and popular online platforms built around Git is GitHub[3]. It contains an enormous collection of more than 20 million repositories and over 5 million users. GitHub is a platform that provides collaboration and code hosting for developers. Users can create both public and private repositories, enabling open and closed source projects. Only with a paid subscription private repositories can be created. Public repositories are free. The repositories on GitHub act as a central mirror of the projects. Every user pushes and pulls their commits to and from the GitHub servers. Merge conflicts have to be resolved locally by the user before pushing to the server. Via this mechanism multiple users can collaborate on the same code base.

Along with the project hosting, GitHub's main feature is their extensive web interface for collaboration. The project's source code can easily be browsed and edited online. It also offers a clear system for creating and discussing issues about the project. If a repository is publicly visible, everyone can create a new issue to e.g. file a bug or request a new feature. When an issue is created it is discussed in its own thread. An issue is closed and archived by a developer if it is either fixed or flagged as irrelevant. GitHub's web interface also has a set of advanced features. Users are able to configure service hooks to run e.g. a continuous integration service when new commits are pushed to GitHub. Also, metrics about the code and the project's collaborators are tracked over time, which gives more insight in the development process.

On GitHub, a pull request is a special case of an issue with the associated code embedded into it. Just like issues, pull requests can be created, discussed and closed. An action specific to pull request is the merge action. With a simple click on a button, GitHub merges the pull request into the target branch. However, this is only possible if there are no conflicts between the pull request and the target branch.

The pull request workflow on GitHub is enabled by implementing the *fork & pull* model. In order to create a pull request, a user has to fork (clone) the main repository first. The user's repository diverges and becomes an actual fork when he commits his changes to his cloned repository. If the developer thinks his changes are useful for the main project he sends a pull request. The fork and pull actions can be executed with a few simple clicks in GitHub's interface.

Pull requests sent by external contributors can have various contents e.g. a bug fix or a new feature. GitHub offers a way to manually categorize the pull requests by assigning labels to them. The overview of open issues and pull requests found on GitHub can be sorted and filtered on several properties like age, number of comments and labels.

Although GitHub recently updated their sorting and filtering mechanism for pull requests [14], they still do not offer a method to easily prioritize the pull requests in a smart way. For example it is not possible to sort pull requests on the likelihood that they are going to be merged. Or if it would be possible to find an ordering of pull requests so that the number of conflicts is minimal, it would save a lot of time.

[3] github.com

2.4 GHTorrent

Because GitHub's platform is so popular for developing open source applications [34], scientists are attracted to it. The software repositories and metadata on GitHub may contain a lot of interesting data. All the data used in this thesis also originates from GitHub. However, it is not trivial to extract these large amounts of data. Although GitHub offers a REST API[4] to fetch various kinds of data, they limit the use at a rate of 5,000 API calls per hour for a single user.

To overcome this limitation and enable mining large portions of GitHub's historical data Gousios et al. [10] introduced the GHTorrent[5] project. The project provides a scalable, queryable, offline mirror of the data offered through the GitHub REST API. By monitoring GitHub's public event stream and storing those events in a MongoDB database, GHTorrent gathers data from GitHub. While GHTorrent saves the raw unstructured JSON responses, it also maintains a structured version of the obtained data in a relational database (MySQL). The relational schema [8] contains entities like *projects*, *commits*, *users*, etc. Since the beginning of the project in 2012, GHTorrent already collected more than 5 terabytes of data.

GHTorrent works in a distributed manner. The communication between the event mirroring and data retrieval phases goes through RabbitMQ messaging queues, so that both phases can be executed on a cluster of machines. It is also possible to help the project by setting up a mirror to ease the load on the GHTorrent servers.

The data that GHTorrent gathers from GitHub is provided back to the community, it is freely accessible for research purposes. Every two months, the project releases the data collected during that period as downloadable archives in an unstructured as well as the extracted structured format. Because downloading and restoring the archives to MongoDB can be very time consuming, GHTorrent also offers an online publicly available version of the data through a SSH tunnel. This allows access to the live databases which reflect the data on GitHub in a near real-time manner. Recently the GHTorrent project introduced a new service, *Lean* GHT*orrent*, which allows researchers to get a slice of the full GHTorrent data set on demand [11].

So GitHub's metadata acts as the main data source for this thesis. However, only a limited set of data is directly obtained through the official GitHub API, instead the GHTorrent project provides us with the largest parts of the metadata.

2.5 Machine Learning

The current deluge of data in the world calls for automated methods of data analysis, which is what machine learning provides [24]. Machine learning is defined as a set of methods that can automatically detect patterns in data. The uncovered patterns can be used to predict future data or to make decisions.

Machine learning is usually divided into two main types: a supervised and an unsupervised learning approach.

[4] developer.github.com/v3
[5] ghtorrent.org

In the supervised learning approach, the goal is to learn a mapping from inputs x to outputs y, given a set of input-output pairs $D = \{(x_i, y_i)\}_{i=1}^{N}$. Here D is called the training set, and N is the number of training examples. In the simplest setting, each training input x_i is a D-dimensional vector of numbers, representing, e.g. the size and age of a pull request. These are called features. However, x_i could be a complex structured object, such as an image, a sentence, an email message, a time series, a molecular shape or a pull request. Similarly the form of the output or response variable can in principle be anything, but most methods assume that y_i is a categorical or nominal variable from some finite set, $y_i \in \{1, \ldots, C\}$ (such as important or unimportant), or that y_i is a real-valued scalar. When y_i is categorical, the problem is known as classification or pattern recognition, and when y_i is real-valued, the problem is known as regression.

There exist a lot of different proven supervised machine learning algorithms that can be used to make data-driven predictions. In general, a machine learning algorithm constructs a model by exploring the training data set. The model can then be used to make predictions about input data that the algorithm sees for the very first time. Rather than following strictly static program instructions, the model acts more like a black-box based on the training data.

The second machine learning type is the unsupervised learning approach. With this approach only inputs are given, $D = \{x_i\}_{i=1}^{N}$, and the goal is to find interesting patterns in the data. This is sometimes called knowledge discovery. This is a much less well-defined problem, since the algorithm is not told what kinds of patterns to look for, and there is no obvious error metric to use (unlike supervised learning, where it is possible to compare the prediction of y for a given x to the observed value).

The supervised machine learning algorithms that are used in our work are: Logistic Regression [15], Naive Bayes [21] and Random Forest [4].

2.6 Bug triage

The process of determining the priority of a set of objects is called triage. A well-known example is the triage process at the emergency department of a hospital. The priority of arriving patients is based on the severity of their condition.

Before pull requests were introduced, bug tracking was something that existed for quite a while. Not surprisingly, there are already several studies that discuss the prioritization of bug reports or issues i.e. bug triage. Several studies have been put into practice by implementing tools that automatically prioritize and/or assign bug reports.

Kanwal et al. [18] applied machine learning classification techniques on historical data from a bug repository that they use to prioritize new bug reports. They explored different features within bug reports to determine which features contribute more towards bug priority classification. Using these features as input for their Support Vector Machine (SVM) algorithm they achieve a recall of 56% and a precision of 54%. They also propose two new measurement units: Nearest False Negatives (NFN) and Nearest False Positives (NFP) which indicate bug reports that are placed into the nearest priority classes. Scores of 75% and 78% are reached for NFN and NFP respectively.

Many of the existing studies to bug triage have a focus on automating the process of assigning a bug report to a particular developer. Since 2005 Anvik et al. [2] have been researching bug repositories to reduce the effort of bug triage. One of their latest studies [3] presents a machine learning approach to create recommenders that assist with the assignment of a developer to a bug report. Using this approach they reach a precision between 70% and 98% and a recall between 1% and 30% for top-1 recommendation over five open source projects.

Xuan et al. [36] also try to provide predictions to assist with the bug triage process. In order to make predictions, they build a developer ranking with a custom algorithm called *developer prioritization* (DP). These DPs are created with a socio-technical approach by feeding the algorithm with bug comments made by developers in bug repositories. Ultimately, the results indicate that DP can improve existing solutions based on SVM with 10% for top-5 recommendation.

A novel hybrid bug triage algorithm approach was introduced by Zhang et al. [38]. It combines a probability model with an experience model to rank all candidate developers for fixing a new bug. For creating the probability model they use a similar socio-technical approach to the one mentioned earlier [36]. The experience model is constructed by taking the number of fixed bugs and fixing cost for each candidate developer to estimate whether the developer has sufficient experience. According to their average F-score of 69% for top-6 recommendation they claim that their hybrid algorithm can effectively recommend the best developer for fixing bugs.

2.7 Pull request usage

Several studies to the usage of pull requests have been done. Gousios et al. [9] state that the majority of the projects on GitHub has only one developer, often with no pull requests at all. Only 14% of the repositories on GitHub are using pull requests. The pull request usage is increasing in absolute numbers, even though the proportion of repositories using pull requests has decreased slightly. In their dataset, most pull requests (84.73%) are eventually merged. Merging pull request can happen in three ways: (1) *Through GitHub facilities*, when there are no conflict, merging can happen with a simple push on a button. (2) *Using Git merge*, more control over the merge can be accomplished by using the offline Git command line tool. (3) *Committing the patch*, the merger creates a new commit with the pull request's changes.

Additionally, they state that the open nature of GitHub's pull requests also lends itself to a variety of usage patterns. Except from basic patch submission, pull requests can be used as a requirements and design discussion tool. In this case, a pull request serves as a discussion board for soliciting the opinions of other developers while a new feature is being implemented. Pull requests can also be used as a progress tracking tool towards the fulfillment of a project release. In GitHub, pull requests can be associated with milestones in the issue tracker.

The findings of Gousios et al. [12] reveal that integrators successfully use pull requests to solicit external contributions. Integrators also use the pull-based model to accommodate

code reviews and discuss new features. Three-quarters of the integrators conduct explicit code reviews on all contributions and they prefer a way of merging that preserves commit metadata.

The work also identifies challenges that arise when using pull requests. Social challenges include motivating contributors to keep working on the project, reaching consensus through the pull request mechanism and explaining reasons for rejection without discouraging contributors. Integrators are also struggling to maintain quality and mention feature isolation and total volume as key technical challenges. Integrators typically need to manage large amounts of contributions requests simultaneously, the work shows that prioritization is a main concern.

2.8 Pull request triage

The topic we investigate in this thesis, prioritization of pull requests, is about which pull requests are more important or need attention first with respect to other pull requests. However, we could not find any existing literature about this. Related to the topic of prioritization, we found several studies that address the assignment of pull requests to reviewers. Thus, the problem of determining which developers are appropriate to review (and eventually merge) specific pull requests, i.e. dividing the open pull requests based on domain among the available developers.

Yu et al. [37] implemented an online service[6] to help project managers to find potential reviewers for pull requests from crowds. Their recommender approach takes advantage of the textual semantic of pull requests and the social relations of developers. The expertise of a reviewer can be learned from his pull request commenting history, developers who have commented on similar pull requests in the past are suitable candidates to review a similar new one. Also the relation between the contributor and the reviewer is taken into account, developers who share more common interests with the contributor are more appropriate to review the contributor's incoming pull request. According to the authors the online system reaches a precision of 74% for top-1 recommendation, and a recall of 71% for top-10 recommendation.

A different approach to reduce the cost of manually selecting the pull request reviewers is proposed by Thongtanunam et al. [35]. They came up with a reviewer recommendation algorithm which determines the file path similarity, called FPS algorithm. Their key assumption is that files that are located in similar file paths would be managed and reviewed by similar code reviewers. The motivation behind this is that in most large systems the directory structure loosely mirrors the system's architecture. The FPS algorithm was tested on three different open source projects and reached an accuracy of 77% for top-5 recommendation.

[6] rrp.trustie.net

Chapter 3

Analysis

In this chapter we discuss the results of a questionnaire among 745 developers about prioritization of their pull requests. To get an idea of how developers and integrators currently handle pull requests in their repositories, we analyzed their responses.

3.1 Survey participants

We had the opportunity to add some additional questions concerning pull request prioritization to an on-going study [12]. The responses of the survey were analyzed by both the initiators of the survey and the author of this thesis.

The participants of the questionnaire were selected by consulting the GHTorrent database (section 2.4). By querying the repositories that have received at least one pull request for each week in the year 2013 (3,400 repositories) it is ensured that the sample group makes effective and large scale use of pull requests. For each repository, the top 3 pull request integrators were extracted i.e. the people that have merged the most pull requests within the repository. The first developer of the top 3 whose email was registered with GitHub received a personal email with the web address of the survey. Several people forwarded or advertised the survey address to others.

Approximately 3,400 people were invited to fill in the questionnaire. Within the first 11 days, a number of 733 complete responses were received. In the end after 16 days, a group of total 745 people answered the two questions about prioritization.

3.2 Questions

As said earlier, we took advantage of an existing study [12]. In collaboration we added two extra questions about prioritization of pull requests to the survey. The questions were worded as follows: (1) *What heuristics do you use when prioritizing pull requests for merging?* (2) *Imagine you frequently have more than 50 pull requests in your inbox. How do you triage them?* The first question was open-ended, the second was a multiple choice question. Results and statistics are presented in the next sections.

3.3 Processing

To give the participants more freedom in giving their answers an open-ended question was included. The downside of an open-ended question is that it is more difficult to perform an analysis on the answers. To say something meaningful about the open-ended question the answers have to be classified. We determined by hand the top 3 heuristics given in each answer. For some of the heuristics we also identified a subcategory e.g. *LIFO*[1] or *FIFO*[2] for the age heuristic.

As part of the questionnaire each participant was asked to fill in the type of work they are in (e.g. the industry, government, academia or open source software) and the name of the repository they mainly handle pull requests for. With the knowledge of the repository name, GHTorrent can be used to retrieve the average number of new pull requests per day in the last 3 months for each repository. By analyzing this information an attempt can be made to find correlations between the given answers and the projects the respondents work on.

3.4 Results

The total number of received responses is 745. More than half of the respondents work in the industry, while only one-fifth work for an open source project. Figure 3.1 shows the distribution of work fields among the participants. Around one-fifth of the respondents (169) say that they do not have a heuristic for prioritizing their pull requests. Most of the time these respondents do not have many outstanding pull request, so in their case prioritization is not necessary. R237: "*I know the project well. We don't have a lot of pull requests, and it's fairly easy to process them all.*"

3.4.1 Prioritization heuristics for merging

The developers who indicated that they prioritize the incoming pull requests use very different approaches. We identified the top 3 heuristics in each of the given answers of the first question, the results can be seen in figure 3.2.

Size/complexity The distinction between *large* pull requests and *complex* pull requests is not always clear and are often correlated. The time it takes to review a pull request is also correlated to these two. Big pull requests tend to be more complex than smaller (and often simpler) pull requests and take more time to review. As the terms short/small/simple and large/complex are used interchangeably by the respondents, we combined them into one heuristic. R473: "*Short simple pull requests are prioritized over larger ones.*" During the classification of the answers, some clear distinct approaches submerged with this heuristic. We divided them into subcategories within the heuristic, these subcategories can be seen in figure 3.3(a). More than two-thirds of the respondents that use this heuristic process the simple (or least time consuming) pull requests first.

[1] Last in, first out.
[2] First in, first out.

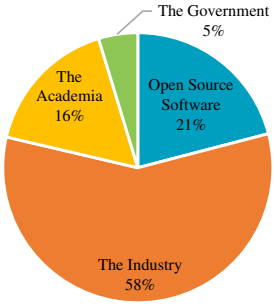


Figure 3.1: Work field distribution. The different work fields of the respondents.

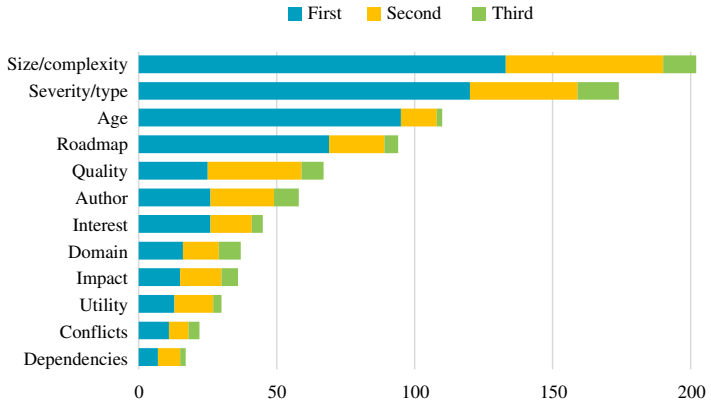


Figure 3.2: Classified prioritization heuristics. The classification of the top 3 heuristics used for prioritizing pull requests given in the answers on question 1.

Severity/type For the same reason we combined *severity* (or criticality) and *type* into one heuristic. In many cases a pull request that contains a critical security bug fix is prioritized over other bug fixes. And in general, bug fixes take precedence over refactoring, new features and documentation additions. R188: "*Bug fixes first, by severity, then enhancements, then new features.*"

Age A large part of the integrators say that they prioritize on the *age*, i.e. the creation time, of the pull requests. Almost three-quarters of the participants that use this heuristic indicate that they use the simple FIFO approach, see figure 3.3(a). R290: "*First in, first out. Nobody, even among owners, should not pull his/her own contribution, unless this has been exposed to community's scrutiny for a week.*"

Roadmap Organized projects plan ahead and determine which features should be completed within the next milestones, they have a so-called *roadmap*. The roadmap heuristic takes the project's current focus into account when prioritizing pull requests, e.g. features in current milestone, predetermined feature importance or time to next release. R406: "*The ones solving issues scheduled for the next release are important.*"

Quality The heuristic that cannot be missed is prioritizing on the *quality* of the pull requests. Is the code understandable and maintainable? Is the code well tested and does it pass the tests? Is the code well documented? Those are some of the questions used to assess the quality of a pull request. R5: "*I prefer pull requests which are well formatted and well thought out. I also prefer pull requests which address bugs over ones which are enhancements.*" In figure 3.3(c) we divided this heuristic into different categories. Almost half of the respondents that use this heuristic prioritize well tested over non-tested code.

Author The heuristic that also makes sense is prioritizing on the *author's* characteristics. Some integrators prefer pull requests of (trusted) contributors who have a positive

history with the project. Other integrators prioritize contributors with a good reputation (or track record) in other projects over complete newcomers, while there are also some integrators that let newcomers take precedence. They do this to stimulate the contributions of newcomers in the future. Figure 3.3(d) shows that the majority of the people that use this heuristic process trusted or reputed contributions first. R261: "*Known contributors get priority.*"

Interest Pull request of open source software hosted on GitHub are public, everybody can view and comment on them. Sometimes heavy discussions on specific pull requests can arise, or a pull request is created for a feature with a lot of feature requests. In those cases the integrators may prioritize these *interesting* or popular pull requests over ordinary pull requests. The personal interest of the project owner(s) is also classified by this heuristic. R271: "*Number of comments on the issue. Number of issues referring to the bug. Basically heat.*"

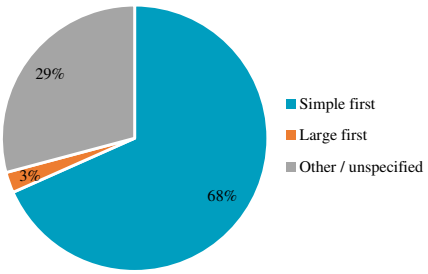
Domain In larger projects, there are often multiple integrators who process the pull request queue. Each integrator has its own *domain* or set of areas of expertise and processes pull requests that affect those areas. This heuristic prioritizes on the integrators domain or whether a pull request affects current development. R467: "*If I am the most obvious developer to review the pull request, I will do it more or less immediately. Otherwise I may let them sit to see if others may want to take it.*"

Impact When using this heuristic, integrators try to determine the *impact* or risk the pull request has on the project. Pull requests may affect the performance of the project or may introduce undesired adverse effects or changes in compatibility. This heuristic is only used by a few developers. R23: "*The code is straightforward and doesn't break existing apps.*"

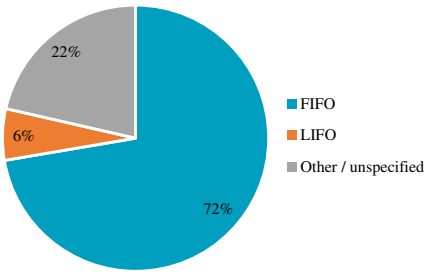
Utility Because everyone can create a pull request for a project, a pull request it is not always *useful* to the project. This can be used in the prioritization. A few integrators say they assess the necessity of the proposed feature and whether the feature is useful for the majority of the users. The assessment happens ad hoc i.e. without a predetermined schedule or planning and is therefore different from the roadmap heuristic. R284: "*Those that seem the most valuable to the project.*"

Conflicts Pull requests that can be merged without *conflicts* might be prioritized over conflicted pull requests. Conflicted pull request take usually more time to review and merge. Some integrators indicate that they investigate whether a pull request conflicts with other pull requests to anticipate on (and possibly reduce) the number of conflicts in future merges. In figure 3.3(e) it can be seen that only one approach stands out: process pull requests with no conflicts first. R591: "*Assessment of what will conflict with other pull requests.*"

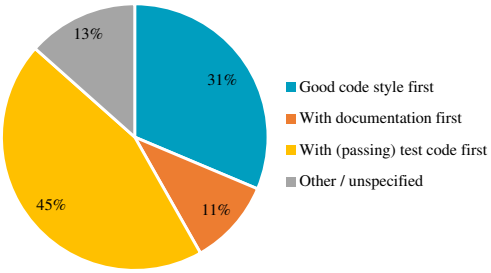
Dependencies Finally, the *dependency* chain can also be used in prioritization. Some pull requests may block further development of other pull requests or even other projects. R69: "*Pull requests that other projects are depending on are prioritized highest.*"



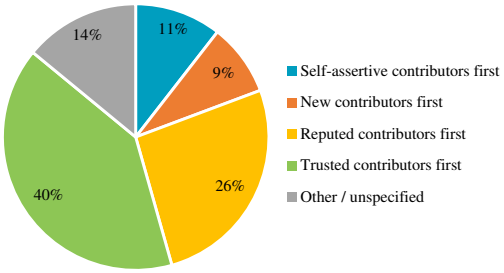
(a) Subcategories of the size heuristic.



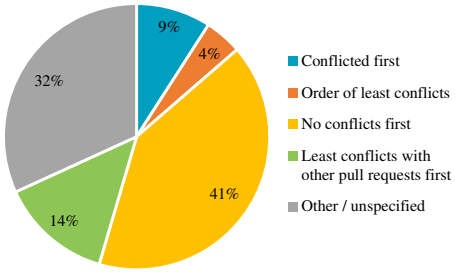
(b) Subcategories of the age heuristic.



(c) Subcategories of the quality heuristic.



(d) Subcategories of the author heuristic.



(e) Subcategories of the conflicts heuristic.

Figure 3.3: Heuristic subcategories. The different subcategories of the several heuristics.

To find distinct groups in the results we first tried to visualize which heuristics are used in combination with each other. The cross correlation can be seen in figure 3.4. There seems to be no clear correlation between the heuristics, the strongest is the domain-author combination with $\rho = 0.14$. Another thing that can be observed is that the age heuristic is negatively correlated with almost all others. This means that the age is often used as a single heuristic.

Some heuristics are more used in specific types of work. The correlation can be seen in figure 3.5. In the industry the quality, roadmap and utility heuristics are more important. The open source world makes more use of the severity and type approaches.

By correlating the heuristics to the average number of pull requests the projects received per day we obtained figure 3.6. It can be seen that some heuristics e.g. conflicts and roadmap are more used if the number of pull requests increases. Additionally, the open source field receives on average more pull requests per day than the other fields.

As a second attempt to discover groups within the results we tried to classify the heuristics into several clusters. Each answer with the top 3 combination of heuristics was fed to a K-means clustering [13] algorithm implemented in R. Using the clustering algorithm we divided the answers into 2 to 10 different clusters. Unfortunately, each time no clear distinct set of clusters was produced. The clusters overlapped each other almost entirely and it seemed that each cluster contained merely a random set of the original data.

3.4.2 Triage methods

The second question was a multiple choice question. The participants were asked to rate 6 statements: (1) *I delegate to developers that are more experienced with the specific subsystem.* (2) *I process them serially.* (3) *I just discard very old pull requests.* (4) *I discard too discussed / controversial pull requests.* (5) *I trust pull requests from reputed pull requesters.* (6) *I assess the technical quality of the pull request.*

The results are shown in figure 3.7. It is clear that the quality is important when triaging pull requests, 45% say that they would always assess the quality. This is particular, because in the previous question the quality was not a very dominant factor. And finally, old or controversial pull requests would be seldom discarded.

3.5 Discussion

From the results it is clear that the majority prefer to process small and simple pull requests over others. In general, small pull requests have less risk and less conflicts, are easier to review, take less time and can be reviewed between other tasks. R622: "*Shorter is better. Small commits are easier to review, and more likely to be merged. Large commits get ignored because it's hard to see what changed.*" There is another simple heuristic that is widely used: FIFO. Together, these two trivial heuristics are used by more than one-third of the respondents. By using these simple methods, developers minimize their time spent on determining on which pull request they should work on next.

	Age	Author	Conflicts	Dependencies	Domain	Impact	Need	Quality	Roadmap	Severity/type	Size/complexity	Utility
Age	-	-0,01	-0,05	0,04	-0,01	-0,06	-0,06	-0,09	-0,11	-0,12	-0,10	-0,07
Author	-0,01	-	-0,02	-0,01	0,14	0,03	0,03	0,12	0,01	-0,04	0,04	-0,01
Conflicts	-0,05	-0,02	-	0,13	0,00	-0,04	-0,04	-0,03	0,01	-0,02	-0,04	0,09
Dependencies	0,04	-0,01	0,13	-	-0,03	-0,03	-0,04	-0,02	-0,03	0,00	0,03	-0,03
Domain	-0,01	0,14	0,00	-0,03	-	-0,02	-0,01	0,06	-0,01	-0,02	0,03	-0,05
Impact	-0,06	0,03	-0,04	-0,03	-0,02	-	0,00	0,06	0,07	-0,07	0,03	0,11
Need	-0,06	0,03	-0,04	-0,04	-0,01	0,00	-	-0,02	-0,05	-0,02	-0,02	-0,05
Quality	-0,09	0,12	-0,03	-0,02	0,06	0,06	-0,02	-	-0,05	0,06	0,04	0,01
Roadmap	-0,11	0,01	0,01	-0,03	-0,01	0,07	-0,05	-0,05	-	0,01	-0,02	0,06
Severity/type	-0,12	-0,04	-0,02	0,00	-0,02	-0,07	-0,02	-0,06	0,01	-	0,02	0,06
Size/complexity	-0,10	0,04	-0,04	0,03	0,03	0,03	-0,02	0,04	-0,02	0,02	-	0,00
Utility	-0,07	-0,01	0,09	-0,03	-0,05	0,11	-0,05	0,01	0,06	0,06	0,00	-

Figure 3.4: Heuristic cross correlation. The table shows how the heuristics are correlated to each other.

	Age	Author	Conflicts	Dependencies	Domain	Impact	Need	Quality	Roadmap	Severity/type	Size/complexity	Utility
Open Source	-0,05	-0,04	-0,01	0,05	-0,01	-0,01	-0,01	-0,03	-0,03	0,09	-0,01	-0,06
The industry	0,00	0,07	0,04	0,02	-0,02	0,02	0,00	0,07	0,05	-0,08	0,01	0,05
The academia	0,02	-0,05	-0,01	-0,07	0,06	0,00	0,04	-0,03	-0,03	0,00	0,02	0,00
The government	0,07	0,01	-0,04	-0,03	-0,05	-0,02	-0,06	-0,05	-0,01	0,01	-0,04	-0,01

Figure 3.5: Work field correlation. The table shows how the work fields are correlated to the heuristics.

	Age	Author	Conflicts	Dependencies	Domain	Impact	Need	Quality	Roadmap	Severity/type	Size/complexity	Utility	Open Source	The industry	The academia	The government
Average new pull requests per day	0,02	-0,01	0,07	-0,01	0,05	0,03	0,03	0,03	0,09	0,01	0,00	-0,04	0,07	-0,01	-0,04	-0,03

Figure 3.6: Newly created pull request correlation. The average new pull requests per day correlated with the used heuristics.

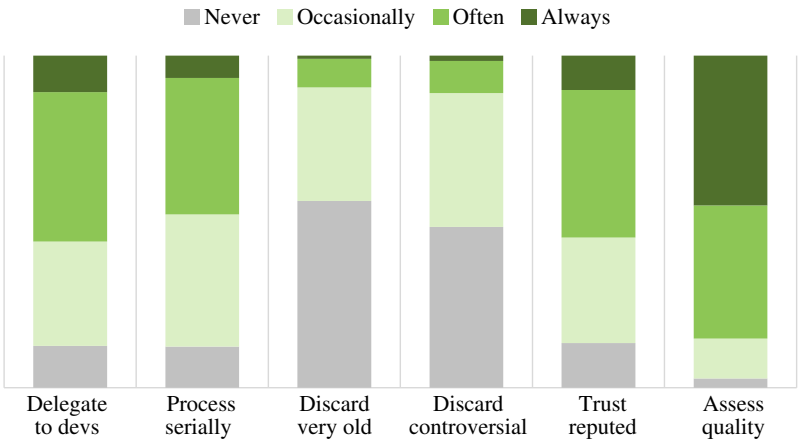


Figure 3.7: Pull request triage method distribution. The distribution of the answers of question 2.

Another insight that can be obtained from the results is that most integrators value (critical) bug fixes over other changes. New bug fixes get immediate attention as opposed to enhancements or new features.

One final matter that we observed is that the quality of a pull request is not very widely used for prioritizing. Only 9% of the respondents use the quality heuristic, 3% use it as their top 1 heuristic. This could be the case, because it is difficult to assess the quality upfront processing the pull request. But, when developers are asked to imagine an inbox of 50 pull request, they overwhelmingly state that they would always assess the quality to triage the pull requests. This significant difference may be caused by people who are not used to a lot of outstanding pull requests and do not realize it is a very time consuming process.

From figure 3.6 we learned that it is difficult to find a connection between the number of pull requests and the prioritization heuristics. It seems that, regardless of the number of received pull requests per day, integrators apply different methods to deal with them.

The discussed questions were part of a larger survey initiated by Gousios et al. [12]. They also classified the answers of the first question, however there was no inter-rater agreement performed between the two studies. This has led to certain differences in the identified heuristics. One of the most obvious differences is that we merged size and complexity into one heuristic, whereas the other analysis treats them as separate categories. Although they mention that the size of the patch is usually related to complexity, they are convinced that a heuristic can be separated into one of the categories. For the reasons described in section 3.4.1 we merged the heuristics into one category. The same holds for the type and the severity (renamed to ‘criticality of fix’) heuristics. And they identified an extra heuristic ‘urgency of feature’ which corresponds partly to the roadmap, interest and utility heuristics in our work. Despite the differences in some of the heuristics, the final observations are in general the same for both studies.

Chapter 4

Modeling

To obtain a prioritization of pull requests we modeled our data using the priority inbox approach [5]. The approach allows us to let the PRioritizer service create a ranking of incoming pull requests. The implicit assumption in this model is that more important pull requests will be acted on sooner than less important pull requests. This is similar to an email inbox. Most people have experienced a long queue in our inbox once and filtered through emails that needed an immediate response versus those that could wait. The filtering that we do naturally is what we will attempt to implement in our model. Pull requests at the top of the list are more *important* and require the developer's immediate *attention*. This section describes which features we extracted from the pull requests and how our training data is constructed.

4.1 Features

To prioritize pull requests, we use a machine algorithm that ranks the incoming pull requests according to a set of features. Most of the features we use are based on the survey answers reported in chapter 3 and are used by integrators to manually sort their pull requests. Table 4.1 shows which features are based on a specific heuristic. An overview of all the features and their statistics can be found in table 4.2.

Heuristic	**Feature(s)**
Age	Age
Author	Accept Rate, Contribution Rate, Core Member
Conflicts	Pairwise conflicts
Interest	Comments, Review Comments
Severity/Type	Contains Fix
Quality	Has Test Code
Size/Complexity	Additions, Deletions, Commits, Files

Table 4.1: Features that are based on a specific heuristic.

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

Table 4.2: Selected features and descriptive statistics for predicting pull request activity. The calculation unit is a pull request. Red plots are histograms with a log scale. Blue plots depict the distribution of the boolean values: false (left) and true (right).

Age The number of minutes between the moment the pull request was opened to the current time. There is a large group of integrators that take a pull request's age into account when manually prioritizing their pull requests.

Contribution Rate The number of commits that is already included in the project *and* authored by the pull requester divided by the total number of commits in the project, i.e. the contribution rate of the pull request author. There are integrators that look at pull requests of known contributors first, their code is of known quality and often faster to review. On the other hand there are integrators that value pull requests of new contributors more than those of known contributors. They think that new contributors are feeling more valued when they receive a fast response on their pull request. Possibly will that feeling encourage them to continue to contribute to the project.

Accept rate The number of accepted (i.e. merged) pull requests authored by the current pull requester divided by the total number of pull requests he/she authored, i.e. the accept rate of the pull request author. A high accept rate can indicate that the author already gained some trust by the integrator, which can result in a higher priority. This feature suffers possibly from accuracy problems, as it is difficult to determine whether a pull request is actually merged or not. A pull request can be merged in different ways [9]. A merge performed via the GitHub interface is easy to detect, but a pull request which is squashed to one commit before the merge is difficult to detect.

Additions The number of lines added by the pull request. Together with the following three features it captures the size of a pull request. A very large group uses the size of the change to manually prioritize their pull requests.

Deletions The number of lines deleted by the pull request.

Commits The number of commits contained by the pull request.

Files The number of files changed by the pull request.

Comments There are different types of comments that can be made regarding a pull request. This feature counts the number of comments on the issue that is automatically created along the pull request. Usually a general discussion about the pull request takes place within the issue. The number of comments might indicate whether a pull request needs more/less attention.

Review Comments Review comments are comments on a portion of the unified diff of a pull request. These are separate from issue comments, which do not reference a portion of the unified diff. This feature counts the number of review comments on a pull request.

Core Member A boolean value indicating whether the author of the pull request is a core member of the project. Some integrators value pull requests of members more than those of non-members.

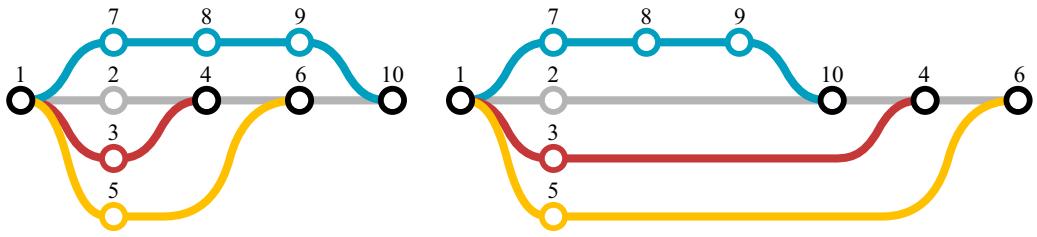
Intra-Branch A boolean value indicating whether the pull request is coming from another branch of the same base project. Creating these kind of pull requests requires write access to the repository, so they can only be created by core members.

Contains Fix A boolean value indicating whether the pull request contains some sort of bug fix rather than e.g. a feature. This is really just an indication as the value is only true if the pull request contains the string "fix" in its title. In general pull requests with a fix are prioritized over other pull requests.

Last Comment Mention A boolean value indicating whether the last comment on a request (regardless of the comment type) contains a mention to a GitHub user. A comment with a mention contains "@username" anywhere in the body of the comment. The mention suggests that the user who is mentioned needs to take action.

Has Test Code A boolean value indicating whether the pull request contains changes to test files. Some projects have an extensive test suite. In these projects it is often expected that new pull requests contain tests for the code they add. The heuristic used for the test code detection is simple, the value of the feature is true if the pull request changes at least one file with "test" or "spec" in its file name.

Target Branch The name of the branch the pull request is targeting in the base repository. It makes sense to use the target branch not as string, but as a categorical variable. Unfortunately the ML algorithm chokes if there is a new branch value that was not



(a) One merge conflict (10). (b) Three merge conflicts (10, 4 and 6).

Figure 4.1: Branch diagram with merge conflicts. Commit pairs $(2,7)$, $(3,8)$ and $(5,9)$ contain conflicting changes.

part of the training data. Therefore this feature is not included in the training data at all, but it is still available for manual sorting.

Pairwise conflicts When the conflicts among other pending pull requests are known, it can be used to determine a merge order in which the occurrences of conflicts is reduced. The actual number of conflicts is not reduced, but they can be concentrated in a particular merge action. Figure 4.1 shows an example of the same set of three pull requests but with different merge orders which result in a different amount of merge conflicts. Pairwise conflicts are only checked among pull requests which target the same branch, but nevertheless the amount of pairs can be very high. In the worst case, i.e. when all pull requests target the same branch, the number of pairs to be checked is $O(n^2)$.

A possible value of this feature could be the number of conflicts between other pending pull requests. However, since checking the pairwise conflicts takes quadratic time, it would take a large amount of time to create the training data. It boils down to $O(n^2 \cdot m)$ where m is the average number of time windows per pull request. Because of this time complexity, we decided to leave this feature out of the machine learning algorithm and only make it available for manual sorting.

RQ1 What features can be used for prioritizing pull requests?

The results of the survey in chapter 3 provided us with several heuristics used for manual prioritization. To get the features that can be used for prioritizing pull requests, we extracted features from those heuristics, which resulted in the feature list presented in table 4.2. For example, many integrators prioritize on the size and age of pull requests, these heuristics are reflected in the Age, Additions, Deletions, Commits and Files features.

4.2 Training data

With the features described in section 4.1 we construct the training data. For reasons discussed earlier we do not include the *target branch* and *pairwise conflicts* features. We

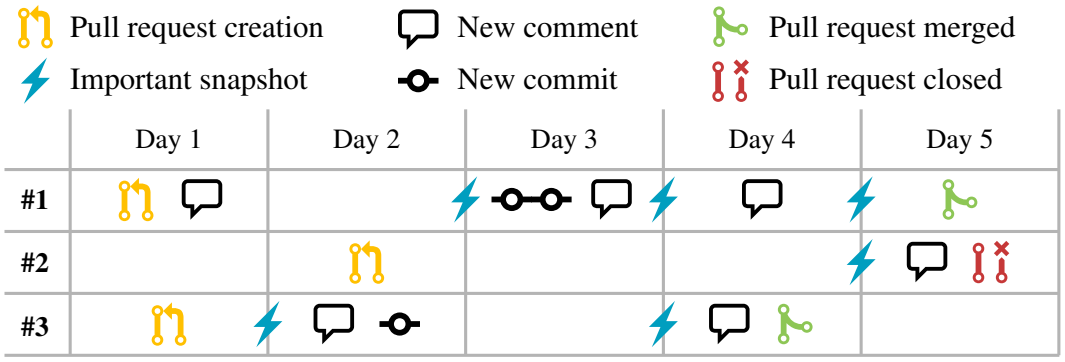


Figure 4.2: Time windows. The snapshots marked with a blue lightning bolt are important because the pull request receives an action the next day.

construct a different training set for every different GitHub project, because every project may employ a different use of the pull-based development model [9]. The different training sets will also result in different prediction models per project, which have to be stored somewhere so that they can be used for making predictions at a later stage.

To train a prediction model so that it recognizes pull requests that need the most attention, we have to feed it with actual examples of pull requests that we consider important and examples of pull requests that we do not consider as important. We define important pull requests as follows: a pull request with a certain state is important when an action follows on that pull request. An action can either be a merge, close, comment or review comment action. All actions are considered equal in terms of importance. To find out what the important states were, we have to dig into the past of previously closed pull requests. We determine the past pull request states by taking a systematic approach of splitting the lifetime of the pull request into time windows. Currently, the time windows have a length of one day. Since it is difficult for developers to find free time in which they can handle pull requests [12], we think it does not make sense to have more fine grained time windows. Also, a more fine grained interval would result in a larger training set, so we decided to predict whether a pull request is acted upon within the next 24 hours. For each window, the state of the pull request at the beginning of the window is calculated by the PRioritizer. A pull request's state consists of its commits, comments and author info at a certain point in time. All this information is extracted from GHTorrent.

Now that a closed pull request can be broken down into a list of consecutive states, we add information about the importance. If a time window encloses one or more actions on the pull request, the state and the beginning of that time window is marked as *important*, as can be seen in figure 4.2. The states of time windows that do not have any actions are not marked as important. The final training data contains all the snapshots of all the closed pull requests. Pull requests that lived longer have more snapshots and appear more often in the training data. Each snapshot is a row in the training data with an extra field indicating its importance.

With the described features and our approach for constructing the training data we have enough information to make an attempt to create a tool that implements this model.

Chapter 5

Implementation

We prove by design how to implement a scalable application that automatically prioritizes pull requests. For this purpose, we implemented a prototype service, the PRioritizer service. It uses the GHTorrent project extensively to extract a project's historical data and feeds it to a machine learning algorithm to prioritize pull requests. As the service requires that the repositories it prioritizes are cloned on the server, a lot of disk space is needed. This is why the service is deployed and being tested on the GHTorrent servers, which have sufficient free space. The prioritized repositories are available via the GHTorrent project website[1]. By using the GHTorrent's pull request event queue, the prioritization of each repository can be kept up-to-date with a small delay. Details about the used technologies and the service's architecture are discussed in the following sections.

[1] ghtorrent.org/prioritizer/

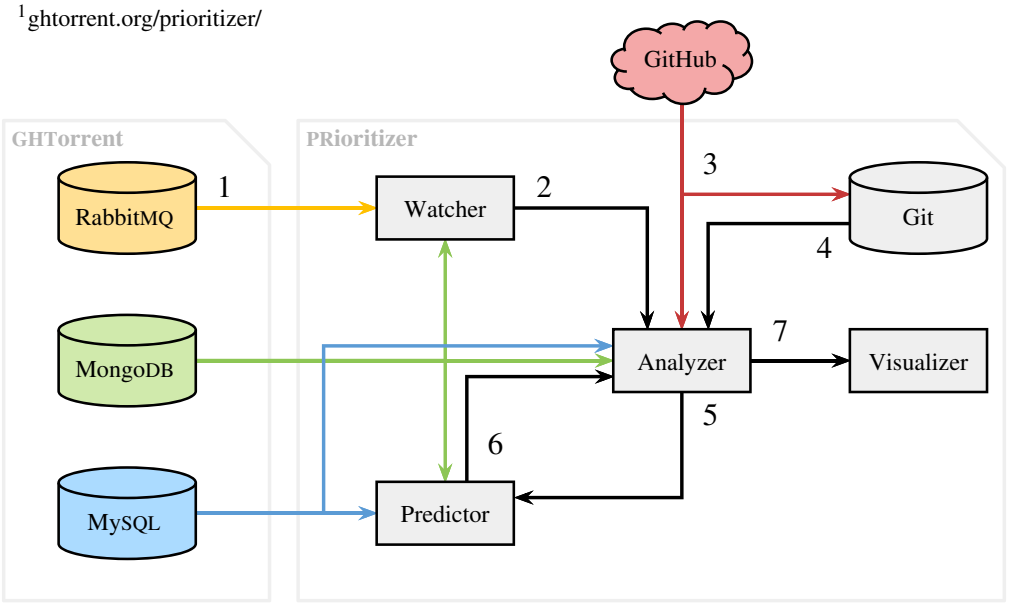


Figure 5.1: Diagram of the architecture. It shows the different data sources and components used by the PRioritizer service. The colored components represent external data sources.

5.1 Architecture

Our implementation of the service consists of several components. Figure 5.1 shows a global overview of the architecture. The PRioritizer service uses two main data sources: GitHub and the GHTorrent project [10]. The latter provides a message queue which can be used to subscribe to pull request events. When such an event arrives the *watcher* component of the PRioritizer is notified (1) and starts prioritizing the project (2). When the *analyzer* gets a prioritization request it fetches the list of open pull requests from GitHub (3). At the same time it fetches the pull request commits to the local Git clone. When the data is fetched the analyzer starts enriching the pull request list with data from the local clone and GHTorrent (4). This step calculates and adds several of the features to the pull requests. The data is now ready to be processed by the *predictor* (5) which gives a certain rank to the pull requests. After the ordered list is returned to the analyzer (6), the output is generated and available for the *visualizer* (7).

5.2 Programming languages

This section gives a short description of the programming languages that are used in the PRioritizer service. All used technologies are cross-platform, open and can be used free of charge.

Scala Large parts of the PRioritizer service are written in the Scala programming language [26]. The high-level Scala language combines features from the object-oriented and functional paradigms and has a very strong static type system. Programs written in Scala are compiled to Java bytecode and run on the Java Virtual Machine (JVM) [23]. Also, Java libraries may be used directly in Scala code and vice versa.

R Next to the Scala code base, we have a small code base written in R. The R language [16] is a programming language and software environment for statistical computing and graphics. It is widely used among statisticians and data miners for data analysis. In our work, the main application of R is the machine learning algorithm of the PRioritizer system. This is because the package repository of R contains a great amount of statistical operations and machine learning algorithms.

5.3 Watcher

At the beginning of the chain we have the watcher, it is implemented in the Scala programming language. The watcher continuously listens to pull request events from GHTorrent via a RabbitMQ message queue. Pull request events are triggered when a pull request is assigned, unassigned, labeled, unlabeled, opened, closed, reopened, or synchronized. The messages in the queue are very lightweight messages, they do not contain any information about the event itself or the repository it belongs to. Instead the messages contain a single object ID that corresponds to a document located in the MongoDB of GHTorrent. This document contains the event itself with all the relevant information, such as the type of the action and

the repository name. The watcher receives the pull request events of all public repositories on GitHub. To distinguish relevant events from non-relevant event, the watcher has to fetch the corresponding event info for every message in the queue. When an incoming event belongs to a repository that is not tracked for prioritization the event is discarded. If an event is received from a tracked repository the analyzer is invoked for that specific repository. Currently, the watcher receives events from GHTorrent at a rate of 3.61 events per second. This rate applies to events that originate from all (relevant and non-relevant) public repositories on GitHub. At the time of writing the watcher prioritizes pull requests for around 550 projects. To increase the rate at which messages can be processed, multiple instances of the watcher component can be started. This way, events from the queue are processed in parallel.

5.4 Analyzer

The component at the center of the service is the analyzer, it is also implemented in Scala. When the analyzer is invoked by the watcher, it is given a repository name. An up-to-date list of open pull requests for that repository is fetched from GitHub through the GitHub API. In parallel the local Git clone is updated with commits of the latest pull requests. After both tasks are done, the list with pull requests is enriched with data from GHTorrent as well as the local Git clone.

The enrichment process adds e.g. information about the pull request author like the *accept rate*, the *contributor rate* and others described in section 4.1. The enrichment process has built-in support for a caching mechanism. Where possible, calculated values are stored in a repository specific cache indexed by the SHA-1 hash of the pull request's tip. It is very likely that a part of the pull requests does not have new commits (i.e. its tip has the same hash as before) in a future run. When a pull request has the same hash as a cache entry, values like the size and mergeability are read from the cache instead of calculating it again, which boosts the speed performance. Features that contain author information (e.g. contributor rate) are not cached, because they change independently of pull request changes.

The local Git clone is used to get information like the mergeability and the *pairwise conflicts* of the pull requests. To gain access to the information enclosed in the local Git clones, we use the JGit library[2], a lightweight, pure Java implementation of the Git version control system. The main benefit of using JGit is the fact that merges can be done in-memory and are not written to the local repository on the disk. Since the merge is not written to the disk, the repository does not have to be reset for the next merge. Because of the in-memory operations, the analyzer is capable of merging pull requests relatively fast, on average 40 merges per second. In practice, the speed is much higher since the use of the caching mechanism which enables incremental pairwise conflicts checking, the rate can go up to around 400 merges per second. The analyzer employs also a way to reduce the number of pull request pairs, it only checks for pairwise conflicts if both pull requests target the same branch. However, because of the $O(n^2)$ nature of the pairwise conflict process, this is still the most time consuming feature of the service.

[2] eclipse.org/jgit

On top of JGit, we use Gitective[3] which makes advanced inspection of Git repositories simpler and easier. It enables the analyzer to check if a pull request contains test code (with the method described in section 4.1) or to count certain properties of a pull request e.g. the number of additions or deletions. This is accomplished by performing a *map and reduce* operation on the commits of a pull request. For example, first the commits are mapped to a value e.g. the number of files a commit contains, then the list of values is reduced to a single value e.g. by calculating the sum. The remaining features e.g. contributor rate or accept rate are calculated by querying GHTorrent. Most of the enrichment phases are executed in parallel for different pull requests.

Some parts of the enrichment information (e.g. the mergeability or number of comments) is also available directly via the GitHub API. This detailed data is present when requesting a single pull request, whereas a request for a list of 100 pull requests lacks this sort of data. This means that if we want this data directly from GitHub we have to create one API request per pull request. When prioritizing pull requests for a large amount of repositories it consumes also a large amount of API requests and reaches GitHub's maximum of 5,000 requests/hour quickly. So, our solution to this problem is to use GHTorrent and the local Git clone instead.

When all blanks are filled in, the enrichment process is done. Some of the calculated values are written to the cache, so that they do not have to be calculated again when the analyzer is invoked in the future.

Before the final output is written to the disk, the predictor is invoked on the enriched pull request list. Details about this process are in the next paragraph. When the list of ordered pull requests is returned back from the predictor, the analyzer completes its process with writing the ordered list of pull requests to the disk in JSON format. The output file will eventually be requested by the visualizer.

5.5 Predictor

The predictor assigns a rank to every pull request, it is partly written in Scala and partly in R. When the predictor is called by the analyzer it receives a list of pull requests. The execution of the predictor is divided into two phases: the training phase and the prediction phase.

The first step in the training phases is checking if there is already an up-to-date prediction model. If this is the case, the predictor skips to the prediction phase. If there is no such model, a prediction model has to be trained on a set of closed pull requests from the past. The training process consists of two steps, first the historical data must be retrieved and secondly the actual model must be trained. The more historical data data is available the better the model can be trained. In general, a specific project needs a few hundred closed pull requests for good prediction model. Information about closed pull request, their comments, the commits they contain and their authors is harvested from GHTorrent. Once all data is fetched, the lifetime of each pull request is divided into time windows to construct the training data as explained in section 4.2. Because all data from GHTorrent is timestamped, it is possible to determine the state of a pull request in each time window. When the states in all

[3] github.com/kevinsawicki/gitective

time windows are calculated the training set is complete. It is then passed to the R program which trains the prediction model. Data communication between R and Scala happens via file exchange in CSV format. The R script reads the input file and calls the machine learning algorithm which learns what features are important for dividing the pull requests into two classifications (important vs. not important). The algorithms and parameters we considered for this purpose are discussed later. The resulting model is saved to the disk for later use when predicting the ranks.

Once an up-to-date prediction model is obtained, the predictor can use it for ordering the list of open pull requests, the second phase. Back in Scala the R program is given the locations of the pull request list and the trained model. Then in R the prediction model is used to calculate for each pull request the probability that it belongs to the *important* cluster. If a pull request is assigned a high probability it means that it is more likely that it receives an action (from the user) soon. We interpret pull requests with a high probability as pull requests that need attention, therefore the ranked list of pull requests is obtained by ordering the list on the importance probability from high to low. The final list is then returned to the analyzer.

5.6 Visualizer

The visualizer is the user interface for the ordered list, it is a web application written in HTML and JavaScript. An example view of the interface can be seen in figure 5.2. To make the development faster and easier we used AngularJS [28], a JavaScript framework. One of the framework's core features is the automatic data binding mechanism between views and controllers. The visualizer itself consists of a static website application which accesses the up-to-date JSON files produced by the analyzer. When a specific repository is requested by the user, the corresponding JSON file containing the prioritized pull requests is parsed and served on the web page.

When the user opens the repository specific page the visualizer shows the list of pull requests automatically sorted on the rank outputted by the predictor. However, the user is also able to sort and filter manually on different fields and features. The GitHub interface for pull requests has only a limited set of fields available for sorting and filtering. It lacks support for sorting on trivial features like the size of a pull request. This is a missed opportunity, because many integrators use it as a manual prioritization feature [12]. As opposed to the GitHub interface, the visualizer interface does support this. And in addition it provides more sort and filter options based on the features from section 4.1 e.g. pairwise conflicts and author properties.

The interface gives the users an overview on which pull requests they have to focus first. However, it is a pure read-only view, to actually interact with a pull request (e.g. merge or close) the user has to go to GitHub's web interface. To make it easier for the user, each pull request shown in the visualizer has a link to its corresponding GitHub version. The list of pull requests shows initially only basic information about the pull requests. If a user wants to know more information about a specific pull request, a pull request can be clicked to reveal more details about the features (see second pull request in figure 5.2).

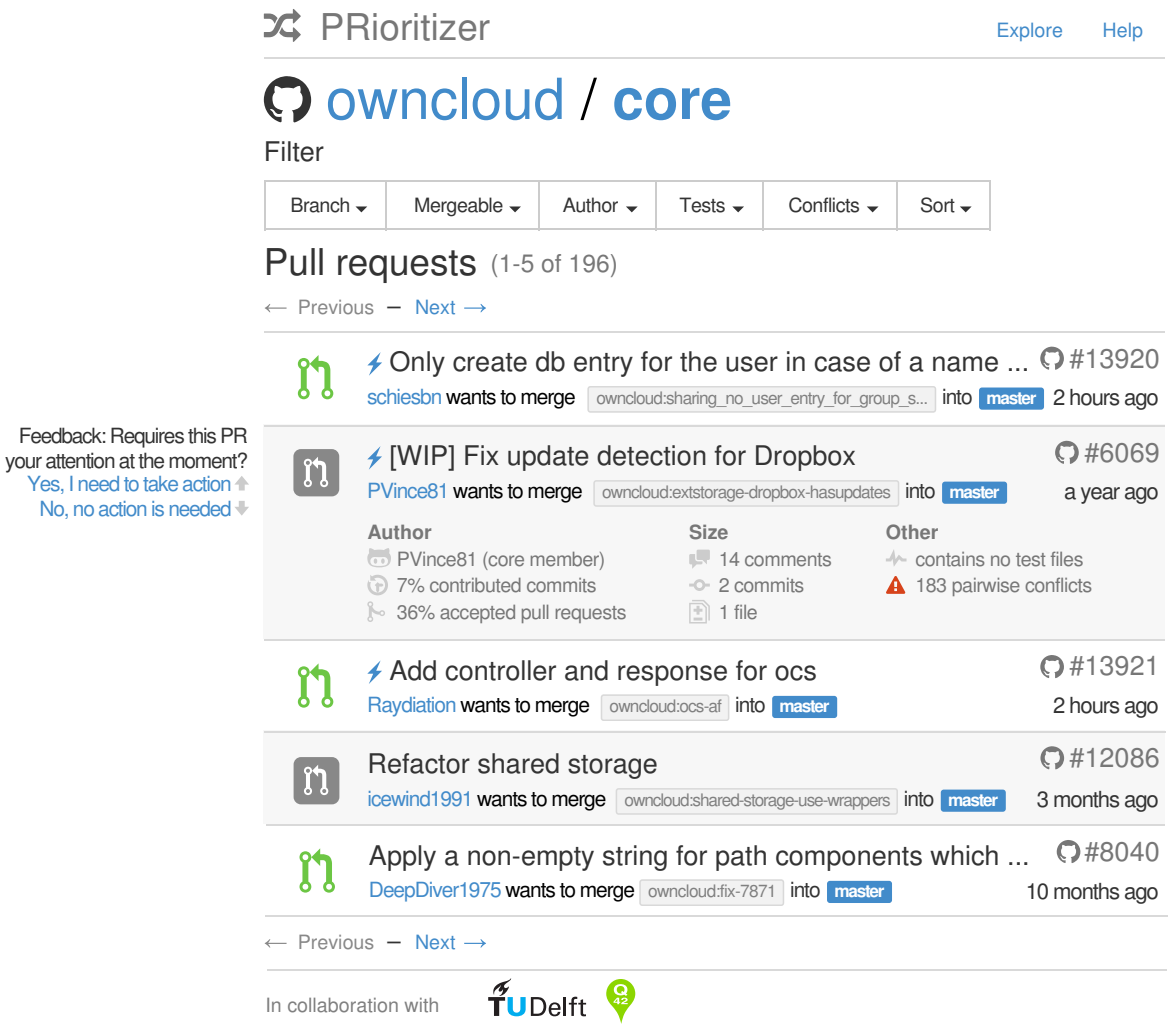


Figure 5.2: The user interface. It shows an ordered list of pull requests that need attention.

A working version, which is also used in our experimental setup, is located on the GHTorrent infrastructure[4].

RQ2 How to implement a scalable system which prioritizes pull requests?

We answer this question by our system design. The architecture of the service is divided in multiple components. Components can be started in parallel to increase the processing rate of the pull requests. We also incorporated a project specific cache that decreases the run time of subsequent runs. The result is that, without great effort, the implementation is capable of prioritizing pull requests in a near real-time manner for at least 550 projects, including most of the largest repositories on GitHub.

[4] ghtorrent.org/prioritizer/

5.7 Extensibility

Large parts of the PRioritizer service are written with extensibility in mind. Especially the data sources used by the service are implemented to be easily interchangeable and extensible. This gives the possibility to add more repository platforms e.g. BitBucket[5] or CodePlex[6].

The data sources are modeled by implementing a `Provider` interface. Classes that implement this `Provider` interface should act as a data provider and manager for a specific data source. Data sources that are currently implemented are: `CacheProvider`, `GitHubProvider`, `GHTorrentProvider` and `JGitProvider`. Every `Provider` provides information of one or more of the following types: repositories (e.g. default branch name), pull requests (e.g. open pull request list) and commits (e.g. total number of commits in repository). For example, the `GitHubProvider` provides an up-to-date list of open pull requests, and the `GHTorrentProvider` can efficiently provide details about commits.

A provider is not only able to provide the PRioritizer with the information mention earlier, but also manages a `Decorator` to 'decorate' or enrich existing objects with information from the provider's own data source. So, if the `GitHubProvider` provides a list of open pull requests then the `CacheProvider` can be instructed to decorate the pull requests with data from the cache. It is possible to chain multiple decorators, so for example after the `CacheProvider` the `GHTorrentProvider` can decorate some extra empty pull request fields from GHTorrent, see listing 5.1. The decoration functionality is loosely based on the decorator pattern [6], a simplified diagram of how this is implemented in the PRioritizer can be seen in figure 5.3. This mechanism is used in the analyzer to enrich the pull request list before predicting the ranks.

The application configuration file describes which data sources and their parameters will be used as provider or as decorator. When the application starts, the configuration file is read and one or more `Provider` instances and decorator chains are constructed and injected into the application according to the dependency injection paradigm [7].

5.8 Usage

The PRioritizer is intended to be used in the code review process. When a developer has time to process one or more pull requests, he/she can take a look at the overview of the PRioritizer. The service then gives the developer suggestions for pull requests to look at. By opening an interesting pull request on the GitHub interface, the developer can decide what action has to be carried out on the pull request. He can comment on or review the pull request on GitHub or in an external application. Finally, if the developer decides to merge or close the pull request, the PRioritizer automatically updates the prioritized list.

The PRioritizer consists of two separate applications, the back-end (watcher, analyzer and predictor) and the front-end (visualizer). They can run separately on different machines and are both implemented with frameworks that support both the Windows as well as the Linux operating system.

[5] bitbucket.org
[6] www.codeplex.com

```scala
class GHTorrentDecorator(base: PullRequestList, val provider: GHTorrentProvider)
    extends Decorator(base) {
  /* The real decorate method updates more than one feature */
  override def decorate(pullRequest: PullRequest): PullRequest = {
    if (!pullRequest.coreMember.isDefined)
      pullRequest.coreMember = Some(isCoreMember(pullRequest.author))
    pullRequest
  }

  /* Execute database query */
  private def isCoreMember(author: String): Boolean =
    queryCoreMember(provider.repository.id, author).firstOption.isDefined

  /* Create typed query with the Slick library */
  private lazy val queryCoreMember = for {
    (repoId, userLogin) <- Parameters[(Int, String)]
    u <- Tables.users
    p <- Tables.projectMembers
    // Join
    if u.id === p.userId
    // Where
    if p.repoId === repoId
    if u.login === userLogin
  } yield u.id
}
```

Listing 5.1: Simplified `decorate` method written in Scala. Queries are built with the database query and access library: Slick (slick.typesafe.com).

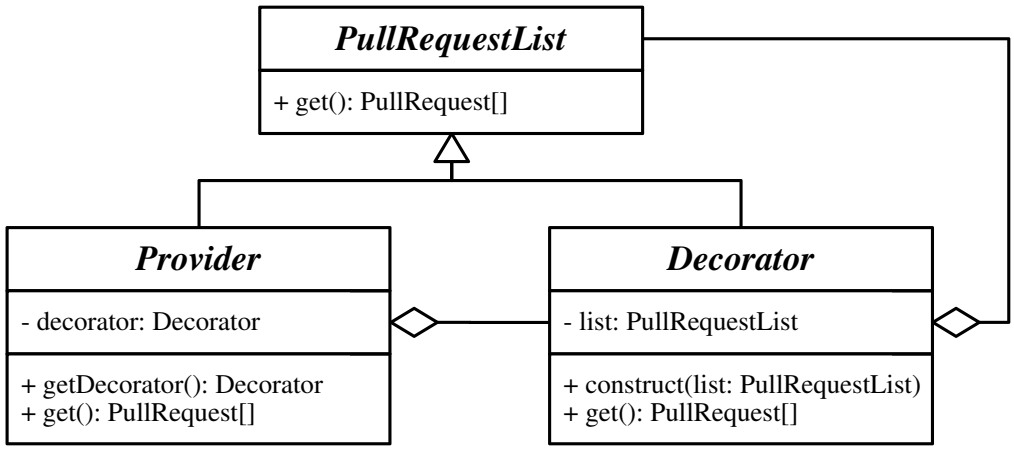


Figure 5.3: The decorator pattern. A UML diagram of how the `Provider` classes are implemented in the analyzer component. The diagram is simplified to only provide or decorate one type of object, namely pull requests.

To run the PRioritizer on a fresh system, the proper run time environments must be installed. The components written in Scala run on the Java Virtual Machine (JVM) and R has its own run time environment which is available for Windows and Linux. The visualizer component consists of a static web application and can run on any regular web server. To install the components their source code must be cloned from the PRioritizer's GitHub page[7]. The four components each have their own Git repository with installation and configuration instructions.

Once the components are in place, the PRioritizer can be started via the command line. The service comes with a command line interface (CLI) tool named `prioritizer` to control the service. It supports three arguments: `start`, `stop` and `attach`, where the latter lets the user attach to the PRioritizer background process to inspect the log messages. A sample of the PRioritizer's log entries can be found in appendix B.

[7] github.com/PRioritizer

Chapter 6

Experimental Setup

Before we measure the performance of our PRioritizer service we have to create an experimental setup. This chapter describes how we tune our service and how we performed a small preliminary qualitative study by inviting developers to use the service on their own projects.

6.1 Machine learning tuning

Our prediction engine is based on a machine learning algorithm. To find out which algorithm to use we measured the performance of several machine learning algorithms. For this purpose we cloned 475 projects from GitHub that act as our test data (see appendix C for the full list of projects). Using the approach described in section 4.2 we constructed the training data for each of the projects. The projects varied in size and pull request count. Three commonly used machine learning algorithms were used on the training data: Logistic Regression [15], Naive Bayes [21] and Random Forest [4]. We ran the three algorithms against each project with a 10-fold cross-validation. The models were trained with 90% of the training data, the remaining 10% was used as test data.

The performance of the algorithms can be measured on different scales e.g. precision, recall and F1 score. A high precision means that the algorithm returned substantially more relevant results than irrelevant. In addition, a high recall means that an algorithm returned most of the relevant results. The F1 score (or simply F-score or F-measure) can be interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 1 and worst score at 0. The F1 score can be calculated as follows:

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

Since the important snapshot data vs. unimportant data is very imbalanced, we decided to tune the balance. By restricting the number of unimportant snapshots in the training data to a multiple of the important snapshots in the training data we obtained more insight in the performance of the algorithms. The different balanced ratios were applied in 10-fold for each project.

As a second attempt to improve the performance, we investigated what kind of predictive power each feature has in a model for all the projects aggregated. By turning some features on or off, the overall predictive power of the model might increase.

To assess the performance of the PRioritizer service, the several components were installed on a single machine with 16GB of RAM memory and a Xeon E5-1607 Quad core CPU running at 3GHz. This machine is also used to run other applications, so despite the powerful hardware, the measured performance might be degraded because of that.

6.2 Evaluation participants

To evaluate the preliminary service we build, we performed a qualitative study to the service's performance. We invited individual integrators to test the service on their own repositories and to fill in a short evaluation form. The selection of the participants was divided into two groups.

The first group of integrators was obtained by recycling a list of people used by a previous questionnaire [12]. Everyone who previously had indicated that they were available for a follow-up, were included in the first group. This gave us a group of 265 integrators. The repositories (244) they manage were also known from the questionnaire and were added to the prioritization service.

The participants received an email containing a direct link to their up-to-date prioritized list of open pull requests and a link to the evaluation form. Unfortunately, the response rate was not high. We received only 9 (3%) forms that were filled in. Because of the low response rate we decided to contact another group of people.

The second group of people consisted of integrators of repositories that have a large amount of open pull requests or a high average new pull requests per day. This list of repositories was obtained by querying GHTorrent, see appendix D for the used queries. After we manually removed some repositories that seemed bogus or were used only for test purposes, 148 repositories remained. Again, we cloned each of these repositories so that the prioritization service built an initial ranking. For each repository we queried GHTorrent to get the top 3 integrators. Without duplicates from the first round, it left us with a group of 304 integrators. Because these repositories have a large amount of pull requests, these integrators would feel the pain of prioritizing pull requests manually the most.

The integrators were asked to take part in a survey (see appendix E) that consisted of 4 open ended questions and 4 sections with a total of 22 even numbered Likert-scale [22] (multiple choice) questions. The Likert-scale questions invited users to rate the usefulness of specific features and the overall service quality while the open ended questions asked the users for missing features or potential improvements. The open ended questions provided the participant with the option to explain their choices in the Likert-scale questions.

To encourage people to respond, we included in our email a performance report[1] of their repository and an opportunity to get a €50 Amazon voucher. This time we received 12 answers (4%), again a low response rate. We also received 10 replies by email, 4 of those contained actual additional feedback. Which brings the total number of responses to 25.

[1] http://ghtorrent.org/pullreq-perf/

Chapter 7

Performance and Evaluation

In this chapter we discuss the performance results of our service. To determine whether the service is able to produce satisfying prioritizations, we validated our service by asking a group of integrators to test and evaluate our service implementation.

7.1 Algorithm performance

In table 7.1 it can be seen how each of the three algorithms perform. Both the Logistic Regression and Naive Bayes algorithms perform poorly on precision, on average 36% and 34% respectively. Their score for accuracy is also not very good, on average 62% and 60% respectively which makes it unusable for our application. The Random Forest algorithm performs less on recall, but achieves a much higher precision and accuracy, which is more important in our case. We want the pull requests that appear at the top of the list to be important (good precision). Important pull requests that are not recognized as such (bad recall) are less of a problem because we only want to show a top 5 of important pull requests as a starting point for the integrators.

The results show that the Random Forests algorithm is the best scoring algorithm. It can predict with relatively high accuracy (86% on average across projects) whether a pull request in a given time frame of its life will be active in the next one. This is an important result for the PRioritizer service, as it gives us the confidence that the Random Forest is a good fit for recommending a pull request prioritization to the user.

As its name already suspects, the Naive Bayes algorithm is based on some naive assumptions that are not necessarily in line with the data. One of the assumptions is that all the features are uncorrelated to each other, but in our case this is not true e.g. the number of additions, files and commits are strongly correlated. Also, compared to the Random Forest algorithm, Naive Bayes is less robust against imbalanced data [19]. It is possible that the Logistic Regression algorithm also suffers from the imbalanced data, however the real reason why the algorithm performs poorly is unknown. A downside of the Random Forest algorithm is that it takes significant more time to construct a model than the other two algorithms.

The Random Forest's average precision, recall and F1 score in table 7.1 are all in the range of $[0.6, 0.7]$. Because the related work described in chapter 2 achieves scores of 0.7 or

Algorithm	5%	Mean	Median	Std	95%	Histogram
Logistic Regression						
Area Under Curve	0.71	0.81	0.81	0.06	0.91	
Accuracy	0.52	0.62	0.62	0.06	0.72	
Precision	0.06	0.36	0.30	0.25	0.88	
Recall	0.66	0.83	0.84	0.09	0.95	
F1	0.12	0.45	0.44	0.21	0.77	
Naive Bayes						
Area Under Curve	0.65	0.75	0.75	0.06	0.86	
Accuracy	0.52	0.60	0.60	0.06	0.69	
Precision	0.06	0.34	0.28	0.23	0.82	
Recall	0.63	0.79	0.80	0.09	0.94	
F1	0.11	0.42	0.41	0.20	0.73	
Random Forest						
Area Under Curve	0.81	0.89	0.89	0.05	0.95	
Accuracy	0.73	0.86	0.87	0.07	0.96	
Precision	0.37	0.66	0.69	0.16	0.90	
Recall	0.36	0.62	0.63	0.15	0.84	
F1	0.38	0.63	0.63	0.15	0.87	

Table 7.1: Comparison of algorithms. Scores and distributions of algorithm performance. The Random Forest algorithm is the best performing algorithm.

Ratio	Logistic Regression			Naive Bayes			Random Forest		
True:False	Prec	Recall	F1	Prec	Recall	F1	Prec	Recall	F1
Unbalanced	0.36	0.83	0.45	0.34	0.79	0.42	0.66	0.62	0.63
1:4	0.36	0.84	0.46	0.34	0.80	0.43	0.60	0.58	0.57
1:3	0.36	0.83	0.45	0.34	0.80	0.43	0.57	0.60	0.57
1:2	0.36	0.84	0.46	0.33	0.80	0.43	0.52	0.65	0.56
1:1	0.35	0.84	0.45	0.33	0.80	0.42	0.46	0.76	0.54

Table 7.2: Comparison of balanced data. The average precision, recall and F1-score for each balanced training set.

higher, it led us believe that there was room for improvement. By restricting the number of unimportant snapshots in the training data to a multiple of the important snapshots in the training data we obtained the averaged results in table 7.2. In case of the Random Forest algorithm, balancing the data makes things worse and it seems to have almost no effect the other two algorithms. A Random Forest model trained with an unbalanced data set is still the best performing option.

When the plot of the Random Forest feature importance in figure 7.1 was obtained

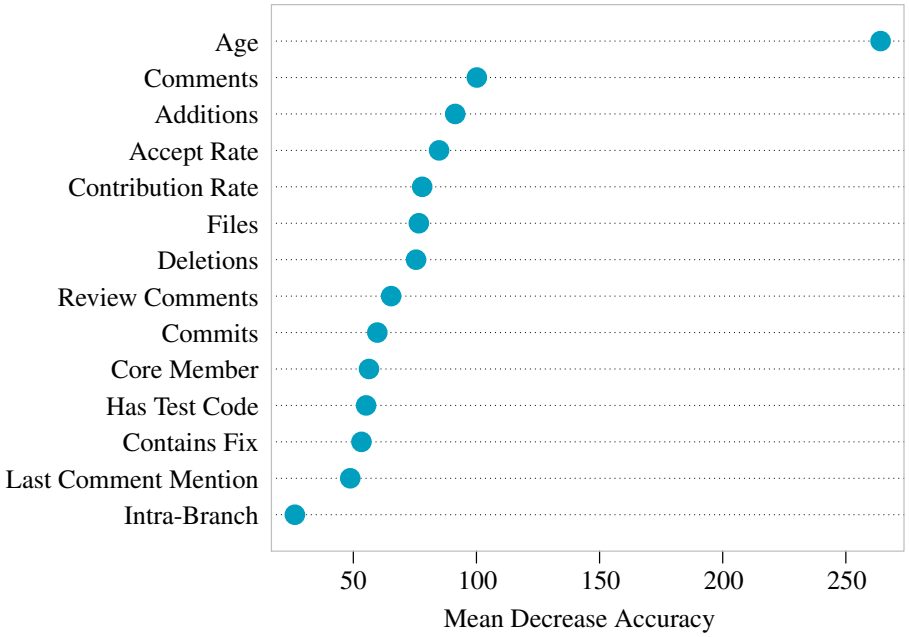


Figure 7.1: Plot of feature importance of the aggregated projects determined by the Random Forest algorithm. The age of the pull request is the dominant factor.

one thing immediately stood out. Of all features that are considered when prioritizing pull requests, it is interesting to see that the age is a very dominant factor. This is probably the case because it is very likely that new pull requests are processed or receive comments within the first day or first few days [9]. Since the age feature is so dominant it could suppress the predictive power of other features and impact the model in a bad way. To see what happens when we give the other features more power, we turned the age feature off. Without the age feature, the results were again worse instead. So it seems the age has a positive effect on the prediction after all.

Based on the results and because this is only a preliminary study to prioritizing pull requests, we decided to improve the model not any further and go forward with the Random Forest algorithm for our prediction model.

7.2 Application performance

Due to the use of a caching-mechanism (see section 5.4) the PRioritizer is able to run incrementally on a set of pull requests. An initial import of a repository, including fetching the Git contents and training data, training the model takes significantly longer than a normal run. The time required for an initial import ranges from a few minutes, for projects with less than a hundred pull requests to a few hours for the biggest of projects. After the initial import, the prioritization time depends on the number of open pull requests: for a medium sized project (30 open pull requests), the processing takes only a few seconds.

On average we found that the service is capable of prioritizing 6.7 pull requests per second and that it performs 40 actual pairwise merges per second, which can go up to a rate of 400 merges per second for projects with a populated cache. The whole process of prioritizing one initialized repository i.e. an incremental run, including updating the local Git repository, takes on average 1 minute and 6 seconds.

When multiple events for big repositories arrive at the same time, a single PRioritizer instance processes them sequentially, which can cause a delay of a few minutes. However, after prioritizing a big repository a single instance always manages to catch up the lost time and processes the events in a real-time manner again after a while. The delay effect can be reduced by starting more instances of the watcher component that process events from the queue in parallel. With the current number of repositories, around 550, that are prioritized there is no need yet for multiple watcher processes.

The front-end of the service, the visualizer component, is not affected by performance issues. Since it is only a static web application all computations required by the visualizer are executed on the client machine. Most web servers should be able to serve a large amount of static web requests per second and are generally not a bottleneck.

7.3 Evaluation

To test our service we invited a group of integrators that maintain one or more open source projects. In addition to analyzing the answers directly we also correlate the answers to the activity of the respondent's project. We define the activity of a project as the average number of pull request actions per day in the last 6 months.

The first questions were about the usefulness of certain features of the service. Table 7.3 and figure 7.2 show the statistics of the given answers. Participants rate insight in the Contribution Rate, Test Code and Size of pull requests as the most useful features. R5: "*The fact you can see how much the author of the pull request did in the past and how his success rate for getting pull requests in. That is really useful information. People that have a track record, will have obviously more chance to have their pull request looked at.*"

However, the main highlight of the service, the Automatic Ordering, is on average neither positively nor negatively rated. This has probably something to do with not knowing the reason why a certain pull request is ranked higher than others, several users indicate that they would like to know this (R6,7,9,17,19,20). R17: "*It can show us the most pressing pull requests. However, it is unclear how this ranking is established, so I'd hope to know why a pull request is considered more urgent then others.*"

Another thing that can be observed from table 7.3 is that the Target Branch and the Pairwise Conflicts features (two manual features) are rated more useful for projects with more activity.

The next part in the survey was about the usability of the service. As can be seen in figure 7.3, the usability has been positively perceived by human evaluators in general. A large majority of 86% (18) state that using the service causes almost no extra overhead. The given answers have a correlation of $\rho = 0.4441$ with the activity of the project.

Feature	Average	Activity correlation
Contribution Rate	0.2143	0.1124
Has Test Code	0.1667	0.0186
Size	0.0952	0.3835
Pairwise Conflicts	0.0476	0.4845
Automatic Ordering	0.0000	−0.0149
Accept Rate	−0.0238	−0.2268
Target Branch	−0.2381	0.6340

Table 7.3: Usefulness of features. The average usefulness of the features and their correlation with the project activity. The answers *strongly disagree, disagree, agree and strongly agree* have the values $-1, -0.5, 0.5, 1$ respectively for calculating the average.

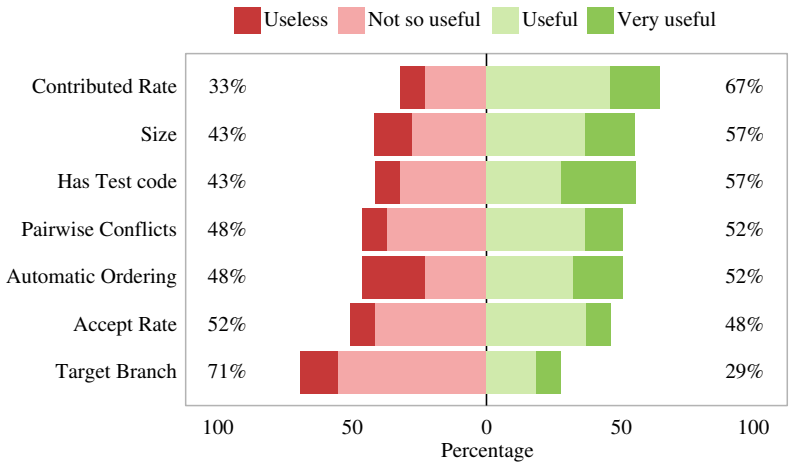


Figure 7.2: User evaluation of individual feature usefulness. Responses of the usefulness of the PRioritizer per feature on a Likert-scale.

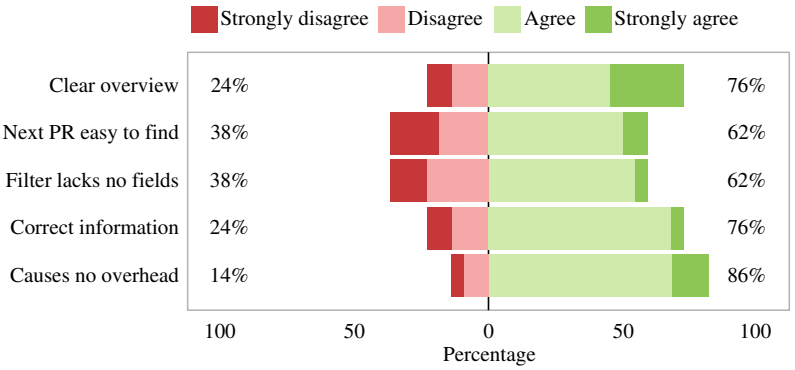


Figure 7.3: User evaluation of the service usefulness. Responses of the usefulness of the PRioritizer service on a Likert-scale.

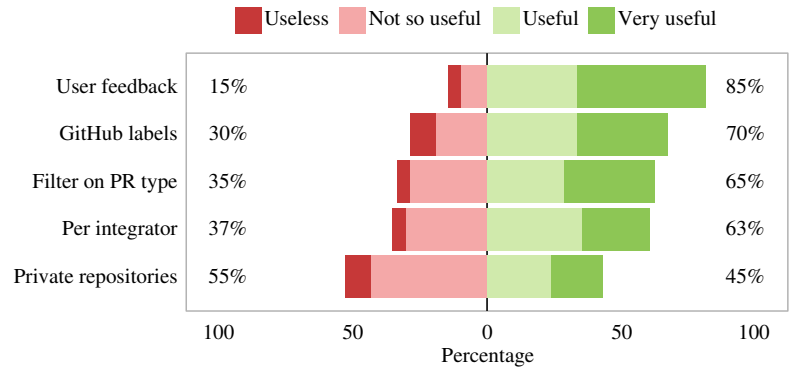


Figure 7.4: User evaluation of future features usefulness. Responses of the usefulness of possible future features of the PRioritizer service on a Likert-scale.

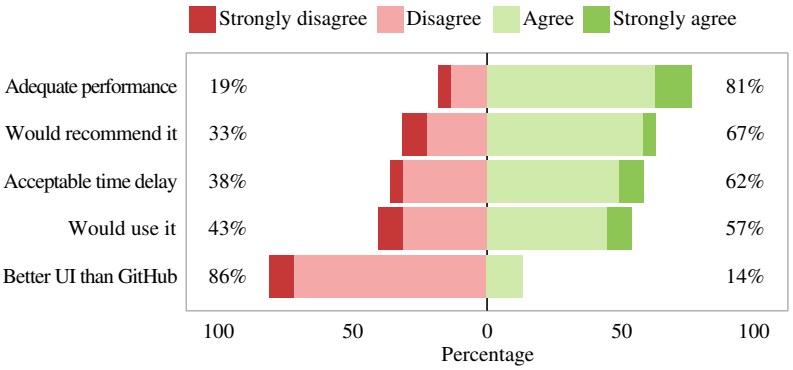


Figure 7.5: User evaluation of the service aspects. Responses of the usefulness of different aspects of the PRioritizer service on a Likert-scale.

Again a large part of the respondents, 76% (16), thinks the prioritized overview of pull requests is clear enough. R1: "*I can see at a glance which pull requests can be merged automatically. For some reason the GitHub pull request interface does not show this, you have to click on a pull request to find out if it can be automatically merged. In one of my projects, pull requests often sit unmerged for a while and have to be rebased, so it's better to know when rebasing is necessary sooner, rather than later.*"

Only 62% (13) of the participants agree that the set of used and presented fields is complete. It seems that the integrators of more active projects tend to disagree more, with a correlation of $\rho = -0.4054$.

Later on in the questionnaire we asked to rate some features for a next version of the prioritization service. Figure 7.4 contains the summary. 71% (15) of the respondents indicated that they want to prioritize according to labels. The correlation between the activity and the lack of label support is $\rho = 0.4443$. R7: "*I like the auto-sorting, but I'd like more control over it. I use labels a lot for things, so for example, 'needs follow-up' is a label that*

means I need to take no action, so I'd love to be able to tell you that."

As said earlier, it is not clear for everyone on what grounds one pull request is higher ranked than others. Almost all respondents (86% or 18) want more control over the automatic ordering in a future version. R19: "*It's very difficult to tell what the default ordering means. This might be an inevitable consequence of machine learning but without some insight into why the results were ordered in a certain way (and maybe the ability to tweak the input weights), the view wasn't helpful.*"

Finally, we asked the respondents if they would use the prioritization service for their project (figure 7.5). A narrow majority of 57% (12) gave a positive answer. With an activity correlation of $\rho = 0.4307$ it seems that integrators with a more active project tend to use it more than those with less active projects. R15: "*I actually don't find it very useful at all. I've never had a problem prioritizing pull requests, we keep the number of open pull requests below 30. Generally, I respond to all pull requests raised on the project as soon as I receive the notification for them.*"

The results are not very conclusive. Due to the small group of respondents, we do not get a clear overview of the general opinion. However, it seems that in its current form the automatic ordering is not adequate. Users want to see why a pull request is considered more important than others and have more control over the learning process. On the other hand the manual sorting and filtering options are positively evaluated. The Pairwise Conflicts, Target Branch and Size are the most popular for projects with a higher activity. Finally, the majority indicated that they will use the service for their project. Especially integrators of projects with a higher activity indicate that they are planning to use it.

RQ3 Is it possible to obtain a useful automatic prioritization of pull requests?

The automatic ordering functionality of the service is neither positively nor negatively rated. This is partly caused by the fact that developers want to understand why a particular pull request appears at the top. However, developers of projects with a higher activity want to use the service, which is a positive signal for the usefulness. Unfortunately, we cannot give a conclusive answer to this question without further research.

Chapter 8

Discussion and Future Work

In this chapter we discuss our implementation and findings. Additionally, some ideas for future work will be discussed.

8.1 Results

The evaluation of the prototype service we developed was mostly positively rated.

The manual sorting and filtering options are positively perceived. Pairwise conflicts, target branch and size are the most popular fields for projects with a higher activity. Although the latter two are fairly trivial features GitHub's interface doesn't support them. This might be the reason why they are rated useful. On the other hand it seems that the integrators of more active projects tend to miss some fields. This is probably caused by the fact that the prioritization service lacks support for GitHub labels and milestones. Those features are often used in projects with a lot of activity to keep things organized.

However, there is a main issue with the PRioritizer service. Many people responded that they would like to have more insight and control over the automatic prioritization function. Users embrace the idea of automatic prioritizing, but they would like to know why certain pull request appear at the top of the list and why others are in the lower parts of the list. They also state that they want to influence the prioritization by giving feedback e.g. by up or down voting specific pull requests.

Developers and integrators struggle with finding free time to handle pull requests. Finding the right pull request to work on next causes overhead and costs extra precious time. Although the service is not implemented in the way they would like to see it yet, our work might help integrators with finding the next pull request to work on, thus saving time. Especially integrators of projects with a higher activity see the potential value of our work.

8.2 Threats to validity

As mentioned before, this is a preliminary study to automatic prioritization of pull requests. Because of this we made some assumptions and decisions throughout our study which might be debatable.

Data sources Most of the data we use is fetched from the GHTorrent project. The PRioritizer service is built in a modular way so that adding and switching to another data source should not be very hard. Despite that, it still depends heavily on GHTorrent and the availability and correctness of its data. This is the case, because currently no real efficient alternatives for GHTorrent exist. The GitHub Archive[1] project is promising, but does not offer a relational data structure, like GHTorrent does, which is needed by the PRioritizer. So for now, the PRioritizer is limited by the data GHTorrent offers, e.g. there is no support yet for the assigned labels of issues and pull requests.

Features We introduced a number of features for the training data set. Some of these features are implemented in a very trivial way. The *Contains Fix* feature is implemented as a boolean value which evaluates to true when the title of the pull request contains the word 'fix'. Another feature, the *Has Test Code* feature, is implemented by looking at the file names of the changed files within pull requests. If one of the file names contains the word 'test' or 'spec', the feature gets the value true. It covers the most used languages[2] on GitHub if they adhere the naming conventions for test files e.g. the file is in a folder named 'test' or has the extension '.spec'. However, these two features are approached in a very simple binary way and may produce false positives. Because this is a preliminary study we wanted to go depth first and left these features as they are.

Machine learning We tested three different machine learning algorithms for predicting pull request importance. With all algorithms parameters left at their default value, the Random Forest algorithm showed the most promising results. The achieved scores are not bad, but also not very good, therefore we went to some extent to improve the scores of the algorithm. Despite the failed efforts we decided to go forward with the current configuration because we were convinced that an accuracy and precision of respectively 86% and 66% were good enough to produce some meaningful results.

Evaluation Upon completion of the PRioritizer service, we invited more than 500 developers to test it on their own repository. Unfortunately we got a response-rate of only 4%, which makes our evaluation less reliable and makes it difficult to give a definitive answer to our main research question. Also, the main highlight of the service, the automatic prioritization, was on average neither positively nor negatively rated. In general the service was positively assessed, the majority indicated they would use the service. Developers working on highly active projects tend to use the service more often than developers of less active projects. However, no hard conclusions can be drawn from the evaluation phase since we only received 25 responses.

It should be possible to create an automated evaluation system, which tracks and compares the predictions made by the PRioritizer and the pull request actions carried out in real life. The need for such a system became clear after we received the poor evaluation response-rate. Since such a system requires great efforts to implement, we decided that further evaluation of the service is out of scope of this initial study.

[1] www.githubarchive.org
[2] githut.info

8.3 Future work

Since this is one of the first studies on the topic of automatic prioritization of pull requests there is plenty of room for improvement.

8.3.1 Further research

Modeling It is hard to create a model that performs well for every repository, the quality of the prediction varies across projects. Maybe it is possible to build some preprocessing phases to determine which model or algorithm may be useful for recommendations.

Another option is to investigate whether it is possible to train a model per integrator instead of one model per repository. With this approach predictions can be made on an individual basis i.e. for one specific developer/integrator with its own domain, but across multiple projects. It will also automate the process of delegating the pull requests among the developers.

A third way to improve the prediction is by learning a different model per pull request action type (comment, review, merge and close). Currently, the model tries to predict that an action will follow in the next day. By training on specific actions it might recognize patterns more easily and enables slightly more insight to the user. Which brings us to the final point.

The evaluation revealed that users want to have more insight and control in the prioritization. In the current implementation it is not possible to extract metadata about the predictions, so a framework should be considered that is capable of this. Also, letting the user influence the model by giving feedback on the outputted prioritization is currently not possible. Random Forest is an algorithm which is not suitable for updating the old model i.e. retrofitting, instead it must be calculated entirely from scratch, which takes a lot of time. An algorithm that is suitable for retrofitting is Logistic Regression, however it did not perform very well in our tests.

Features The current feature set for the training data may be extended. By researching and adding more representative features that are aimed at contents of a pull request e.g. code and domain, the predictions of the machine learning algorithm may improve significant. The user can be given more control by letting him/her decide which features they want to select to account for variations of pull request handling practices.

It is also possible to improve the current features. Since many integrators prioritize on the severity and type of pull requests it may pay off to improve the *Contains Fix* feature. The body of the pull request can be searched for more words and their frequency to give an indication of the type. In addition if a pull request has a reference to a GitHub issue it might have a higher probability of being a fix. The value should be changed from a boolean to a floating point number to indicate the probability or criticality of a fix. In a similar way it should be possible to improve the *Has Test Code* feature.

8.3.2 Further development

Scale The PRioritizer service watches 563 projects which are prioritized on one machine with a small delay. To keep the delay from increasing when more repositories are added, more processes have to be executed in parallel. In the current implementation the different stages of prioritizing a single repository is already parallelized. However, the message queue between the watcher and the analyzer is still processed sequentially. By starting multiple analyzer instances on a cluster of machines it should be possible to process the message queue in parallel, but we did not thoroughly test this yet.

Another scalability issue is the check for conflicts among pull request pairs. This check is the most time consuming operation of the service, because the complexity is quadratic. There are certainly more clever ways and optimizations possible to improve this process. One idea is to build some sort of a tree structure which represents the commits of the pull requests. It then may be easy to reason about and propagate conflicts to commits of other pull requests without actually merging them. A second idea is to use a merge algorithm that stops when the first conflict is found in a merge. Normal algorithms continue to try to merge the rest of the file(s).

User interface The front-end of the PRioritizer service is implemented in HTML, CSS and JavaScript. For the development of the interface we did not consult with a usability expert and implemented it ourselves. Since we are no experts in this area, it is highly possible that the interface lacks a certain degree of usability. It should be possible to increase the usability by improving the interface e.g. by adding visual cues between conflicted pull requests or redesigning the filter options.

Chapter 9

Conclusion

This chapter gives an overview of our findings and summarizes our conclusions. We also mention the contributions of this thesis.

9.1 Conclusions

In this section we state our conclusions by answering the research questions.

RQ1 What features can be used for prioritizing pull requests?

We learned from the extensive analysis of the survey in chapter 3 how integrators prioritize their pull requests manually. In the analysis we identified 12 different heuristics that are used for prioritizing. A large part of the integrators apply simple heuristics like processing the pull requests in the order they were created (FIFO) or by processing the smallest or simplest pull requests first.

Most of these manual prioritization techniques use one or more features of the pull requests to prioritize on. For example the FIFO approach uses the *age* of a pull request. To get a suitable list of features that can be used for automatic prioritization, we extracted the underlying features from the heuristics. We successfully extracted 14 features, which are listed in 4.2.

RQ2 How to implement a scalable system which prioritizes pull requests?

We answer this question with a *prove-by-design*. Chapter 5 describes thoroughly how we designed an application that is capable of prioritizing pull requests. The implementation consists of four components: (1) the *watcher*, which gets notified when a repository needs to be prioritized (2) the *analyzer*, which is responsible for collecting data about the pull requests (3) the *predictor*, which gives the pull request a rank based on its data and (4) the *visualizer*, which displays the ordered pull request list in a clear overview.

The watcher component can be started in parallel to increase the processing rate of the pull requests. In his turn, it invokes other components that also run in parallel. The analyzer implements a caching mechanism that enables incremental prioritization and decreases the

run time of subsequent runs. Without great effort, our PRioritizer service is capable of prioritizing pull requests in a near real-time manner for hundreds of projects, including most of the largest repositories on GitHub.

RQ3 Is it possible to obtain a useful automatic prioritization of pull requests?

In section 6.1 we presented the prediction performance of our chosen machine learning algorithm, Random Forest. Per pull request the algorithm calculates the probability of belonging to the important pull request cluster. The average scores for accuracy, precision, recall and F1 are respectively 86%, 66%, 62% and 63%. Although we believe that it is possible to achieve better scores, we find the results promising.

To get an idea of what integrators think of the service, we sent out an evaluation survey. Our survey did not produce the expected number of responses making it hard to draw a clear conclusion. The participants that tried out the developed service on their own repository are on average neither positively nor negatively tuned about the automatic ordering. Most of them want to have more insight and control over the prioritization. Ultimately, the service as a whole was rated positively and integrators of projects with a higher activity say that they would use the service. This indicates we are on the right track, but there is still a lot to be done. We are inclined to answer the question positively, but further research is needed to give a definitive answer to this research question.

9.2 Contributions

In our work we have made the following contributions.

- We have explored and identified the different heuristics used for manual prioritization of pull requests.
- We have presented a data model consisting of time windows, which captures the importance of a pull request's state.
- We have designed and developed a service which is capable of watching and prioritizing a project's pull requests automatically when they are created.
- We have presented one of the first studies, which investigates the topic of automatically prioritizing pull requests. It may form the basis for new research to this topic.

Bibliography

[1] Brian de Alwis and Jonathan Sillito. “Why are software projects moving from centralized to decentralized version control systems?” In: *Cooperative and Human Aspects on Software Engineering, 2009. CHASE ’09. ICSE Workshop on*. May 2009, pp. 36–39.

[2] John Anvik, Lyndon Hiew, and Gail C. Murphy. “Coping with an Open Bug Repository”. In: *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*. eclipse ’05. San Diego, California: ACM, 2005, pp. 35–39. ISBN: 1-59593-342-5.

[3] John Anvik and Gail C. Murphy. “Reducing the Effort of Bug Report Triage: Recommenders for Development-oriented Decisions”. In: *ACM Trans. Softw. Eng. Methodol.* 20.3 (Aug. 2011), 10:1–10:35. ISSN: 1049-331X.

[4] Leo Breiman. “Random Forests”. English. In: *Machine Learning* 45.1 (2001), pp. 5–32. ISSN: 0885-6125.

[5] Drew Conway and John White. *Machine learning for hackers*. O’Reilly Media, Inc., 2012, pp. 93–125.

[6] James W Cooper. “The design patterns Java companion”. In: (1998), pp. 103–110.

[7] Martin Fowler. *Inversion of control containers and the dependency injection pattern*. Jan. 2004. URL: `http://www.martinfowler.com/articles/injection.html` (visited on 02/27/2015).

[8] Georgios Gousios. “The GHTorrent dataset and tool suite”. In: *MSR ’13: Proceedings of the 10th Working Conference on Mining Software Repositories*. MSR ’13. San Francisco, CA: IEEE Press, May 2013, pp. 233–236. ISBN: 978-1-4673-2936-1.

[9] Georgios Gousios, Martin Pinzger, and Arie van Deursen. “An Exploratory Study of the Pull-based Software Development Model”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 345–355. ISBN: 978-1-4503-2756-5.

[10] Georgios Gousios and Diomidis Spinellis. “GHTorrent: GitHub’s Data from a Firehose”. In: *MSR ’12: Proceedings of the 9th Working Conference on Mining Software Repositories*. Ed. by Michael W. Godfrey and Jim Whitehead. Zurich, Switzerland: IEEE, June 2012, pp. 12–21.

[11] Georgios Gousios et al. “Lean GHTorrent: GitHub Data on Demand”. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: ACM, 2014, pp. 384–387. ISBN: 978-1-4503-2863-0.

[12] Georgios Gousios et al. “Work Practices and Challenges in Pull-Based Development: The Integrator’s Perspective”. In: *Proceedings of the 37th International Conference on Software Engineering*. ICSE 2015. To appear. Florence, Italy, 2015.

[13] John A. Hartigan. *Clustering Algorithms*. 99th. New York, NY, USA: John Wiley & Sons, Inc., 1975. ISBN: 047135645X.

[14] Zach Holman. *The New GitHub Issues*. 2014. URL: `https://github.com/blog/1866-the-new-github-issues` (visited on 10/29/2014).

[15] David W. Hosmer and Stanley Lemeshow. “Introduction to the Logistic Regression Model”. In: *Applied Logistic Regression*. John Wiley & Sons, Inc., 2005, pp. 1–30. ISBN: 9780471722144.

[16] Ross Ihaka and Robert Gentleman. “R: A Language for Data Analysis and Graphics”. In: *Journal of Computational and Graphical Statistics* 5.3 (1996), pp. 299–314.

[17] GitHub Inc. *Using pull requests*. 2014. URL: `https://help.github.com/articles/using-pull-requests` (visited on 10/27/2014).

[18] Jaweria Kanwal and Onaiza Maqbool. “Bug Prioritization to Facilitate Bug Report Triage”. English. In: *Journal of Computer Science and Technology* 27.2 (2012), pp. 397–412. ISSN: 1000-9000.

[19] T.M. Khoshgoftaar, M. Golawala, and J. Van Hulse. “An Empirical Study of Learning from Imbalanced Data Using Random Forest”. In: *Tools with Artificial Intelligence, 2007. ICTAI 2007. 19th IEEE International Conference on*. Vol. 2. Oct. 2007, pp. 310–317.

[20] Daniel Kuhn. “Distributed Version Control Systems”. In: (July 2010).

[21] David D. Lewis. “Naive (Bayes) at Forty: The Independence Assumption in Information Retrieval”. In: *Proceedings of the 10th European Conference on Machine Learning*. ECML ’98. London, UK, UK: Springer-Verlag, 1998, pp. 4–15. ISBN: 3-540-64417-2.

[22] Rensis Likert. “A Technique for the Measurement of Attitudes”. In: *Archives of Psychology* 22.140 (1932).

[23] Tim Lindholm et al. *The Java virtual machine specification*. Java SE 8. Pearson Education, July 2014. ISBN: 013390590X.

[24] Kevin P Murphy. *Machine Learning: A Probabilistic Perspective*. MIT press, July 2012. ISBN: 978-0-262-01802-9.

[25] Kıvanç Muşlu et al. “Transition from Centralized to Decentralized Version Control Systems: A Case Study on Reasons, Barriers, and Outcomes”. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 334–344. ISBN: 978-1-4503-2756-5.

[26] Martin Odersky et al. *An Overview of the Scala Programming Language (2. Edition)*. Tech. rep. 2006.

[27] C. Rodriguez-Bustos and J. Aponte. “How Distributed Version Control Systems impact open source software projects”. In: *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. June 2012, pp. 36–39.

[28] Shyam Seshadri and Brad Green. *AngularJS: Up and Running: Enhanced Productivity with Structured Web Apps*. O’Reilly Media, Inc., 2014.

[29] Ian Skerrett. *The Open Source Developer Report*. Eclipse Community Survey 2009. The Eclipse Foundation, May 2009.

[30] Ian Skerrett. *The Open Source Developer Report*. Eclipse Community Survey 2010. The Eclipse Foundation, June 2010.

[31] Ian Skerrett. *The Open Source Developer Report*. Eclipse Community Survey 2011. The Eclipse Foundation, June 2011.

[32] Ian Skerrett. *The Open Source Developer Report*. Eclipse Community Survey 2012. The Eclipse Foundation, June 2012.

[33] Ian Skerrett. *The Open Source Developer Report*. Eclipse Community Survey 2013. The Eclipse Foundation, June 2013.

[34] Ian Skerrett. *The Open Source Developer Report*. Eclipse Community Survey 2014. The Eclipse Foundation, June 2014.

[35] Patanamon Thongtanunam et al. “Improving Code Review Effectiveness Through Reviewer Recommendations”. In: *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*. CHASE 2014. Hyderabad, India: ACM, 2014, pp. 119–122. ISBN: 978-1-4503-2860-9.

[36] Jifeng Xuan et al. “Developer prioritization in bug repositories”. In: *Software Engineering (ICSE), 2012 34th International Conference on*. June 2012, pp. 25–35.

[37] Yue Yu et al. “Reviewer Recommender of Pull-Requests in GitHub”. In: *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. Sept. 2014, pp. 609–612.

[38] Tao Zhang and Byungjeong Lee. “A Hybrid Bug Triage Algorithm for Developer Recommendation”. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. SAC ’13. Coimbra, Portugal: ACM, 2013, pp. 1088–1094. ISBN: 978-1-4503-1656-9.

Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

API Application programming interface. An API specifies how to communicate between two components. Many online services e.g. GitHub have a public API which can be used to retrieve or send data to that service. Often APIs are used to assist with sharing and integrating data of different applications.

DVCS Distributed version control system. A DVCS allows many software developers to work on the same project without requiring them to share a common network. Distributed revision control takes a peer-to-peer approach to version control, as opposed to the client-server approach of centralized systems. Rather than a single, central repository on which clients synchronize, each developer's working copy of the code base is a complete repository. A common DVCS is Git.

CVCS Centralized version control system. A CVCS allows many software developers to work on the same project. It is based on the idea that there is a single *central* copy of the project on a server. Developers will commit their changes to this central copy. Each developer's working copy is equal to the latest revision of the project. Common CVCSs are CVS, Subversion (SVN).

FIFO First in, first out. It is a method for organizing and manipulating a data queue, where the oldest (first) entry is processed first.

Fork & pull model According to GitHub [17]: "The fork & pull model lets anyone fork an existing repository and push changes to their personal fork without requiring access be granted to the source repository. The changes must then be pulled into the source repository by the project maintainer. This model reduces the amount of friction for new contributors and is popular with open source projects because it allows people to work independently without upfront coordination."

GHTorrent The GHTorrent[1] project is a scalable, queriable, offline mirror of data offered through the GitHub REST API. The data is offered in two formats: a relational

[1] ghtorrent.org

structured (MySQL) and a raw flat format (MongoDB). Currently (Apr 2015), it stores around 6.5 TB of data. See section 2.4 for more information about GHTorrent.

Git Git[2] is a common free and open source DVCS that is designed to handle everything from small to very large projects with speed and efficiency.

GitHub GitHub[3] is an online project hosting using Git. It includes a source-code browser, in-line editing, wikis and ticketing. Public open-source repositories can be hosted for free. With GitHub developers can easily share and review their code. The code and meta-data on GitHub is index by the GHTorrent project and is extensively used in this thesis. More information about GitHub can be found in section 2.3.

Integrator A developer who is responsible for reviewing and merging pull requests.

JGit JGit[4] is a library implementing the Git version control system. Although the library is written purely in Java, it can be used in a Scala application.

JVM Java virtual machine [23]. A platform independent environment to execute Java bytecode. The well known programming language Java compiles to Java bytecode, but there are more languages e.g. Scala that also compile to Java bytecode and therefore also run on the JVM.

JSON JavaScript Object Notation. It is an open standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs. It is used primarily to transmit data between a server and web application, as an alternative to XML.

LIFO Last in, first out. It is a method for organizing and manipulating a data queue, where the newest (last) entry is processed first.

Machine learning Machine learning [24] is defined as a set of methods that can automatically detect patterns in data. The uncovered patterns can be used to predict future data or to make decisions.

Merge conflict Version control systems support branching and merging of code. Usually a new branch is created when a developer wants to work on a new feature in isolation of other parallel changes. When the branch is merged back into the main branch, it is possible that the main branch received some changes on the same spots as the feature branch. In such cases a merge conflict occurs. To resolve the merge conflict a developer has to manually review and change the lines of code that were altered in both branches.

Pairwise conflict A merge conflict between two pull requests. By exploring the pairwise conflicts it is possible to reduce the number of merge conflicts. See section 4.1 for more information.

[2] git-scm.com
[3] github.com
[4] eclipse.org/jgit

PRioritizer The service that was developed for this thesis. It automatically prioritizes pull requests when a new pull request is created in one of the repositories that is being watched.

Precision A measure of relevance for machine learning algorithms. A high precision means that the algorithm returned substantially more relevant results than irrelevant.

Pull request The use of a DVCS, enables a workflow method involving pull requests. Everyone with reading permissions may clone a project to a local repository. Changes can then be pushed to this offline fork. The author of these changes can *request* that his changes should be *pulled* back into the project's main repository. This is a so-called pull request. The core developers decide eventually to accept (merge) or reject the pull request. More information on pull requests can be found in section 2.2.

R R [16] is a programming language and software environment for statistical computing and graphics. It is widely used among statisticians and data miners for data analysis. In this thesis R is used for running the machine learning algorithms.

Random Forest Random Forest [4] is a machine learning algorithm for classification. The algorithm generates a set of decision trees with random decision variables. Since the trees are generated at random, most trees have no predictive power at all. Along with the bad models, a few really good decision trees are generated.

When a prediction is made, each of the trees (good and bad ones) produce a prediction for the classification. The predictions of the bad models are all over the place and cancel each other out. The predictions of the minority of trees that are good top that noise and yield a good prediction.

Recall A measure of relevance for machine learning algorithms. A high recall means that an algorithm returned most of the relevant results.

Roadmap Organized projects plan ahead and determine which features should be completed within the next milestones, this is called a roadmap.

Scala The Scala [26] programming language is a high-level language, which combines features from the object-oriented and functional paradigms and has a very strong static type system. Programs written in Scala are compiled to Java bytecode and run on the JVM.

VCS Version control system. A management system for the changes to documents, computer programs and other collections of information. Each revision is associated with a timestamp and the person making the change. Revision control allows for the ability to compare different revisions, revert a document to a previous revision and, with some types of files, merge documents of different revisions.

Appendix B

Service log

```
2015-02-06 12:49:13:415 [INFO] Watcher - New event - ID: 2563303860
2015-02-06 12:49:13:416 [INFO] Watcher - Database - Timestamp: 2015-02-06 12:45:25.000
2015-02-06 12:49:13:416 [INFO] Watcher - Database - Action: closed
2015-02-06 12:49:13:416 [INFO] Watcher - Database - Number: 13942
2015-02-06 12:49:13:416 [INFO] Watcher - Database - Repository: owncloud/core
2015-02-06 12:49:13:426 [INFO] Watcher - Prioritizing - Start process
2015-02-06 12:49:13:775 [INFO] Analyzer - Setup - Done
2015-02-06 12:49:13:845 [INFO] Analyzer - Fetch - Start
2015-02-06 12:49:14:284 [INFO] Trainer - Skip - Already recently updated
2015-02-06 12:49:23:696 [INFO] Analyzer - Fetch - End
2015-02-06 12:49:23:696 [INFO] Analyzer - Single - Start
2015-02-06 12:49:34:137 [INFO] Progress - 10% (20/200)
2015-02-06 12:49:40:165 [INFO] Progress - 20% (40/200)
2015-02-06 12:49:47:149 [INFO] Progress - 30% (60/200)
2015-02-06 12:49:53:426 [INFO] Progress - 40% (80/200)
2015-02-06 12:50:01:583 [INFO] Progress - 50% (100/200)
2015-02-06 12:50:09:981 [INFO] Progress - 60% (120/200)
2015-02-06 12:50:16:850 [INFO] Progress - 70% (140/200)
2015-02-06 12:50:21:723 [INFO] Progress - 80% (160/200)
2015-02-06 12:50:32:606 [INFO] Progress - 90% (180/200)
2015-02-06 12:50:42:758 [INFO] Progress - 100% (200/200)
2015-02-06 12:50:42:758 [INFO] Analyzer - Single - End
2015-02-06 12:50:42:758 [INFO] Analyzer - Total - Start
2015-02-06 12:50:43:160 [INFO] Predictor - Prediction - Start
2015-02-06 12:50:43:992 [INFO] Predictor - Prediction - End
2015-02-06 12:50:44:016 [INFO] Analyzer - Total - End
2015-02-06 12:50:44:016 [INFO] Analyzer - Pairwise - Start
2015-02-06 12:50:59:151 [INFO] Progress - 10% (1874/18736)
2015-02-06 12:51:03:528 [INFO] Progress - 20% (3748/18736)
2015-02-06 12:51:09:708 [INFO] Progress - 30% (5621/18736)
2015-02-06 12:51:17:023 [INFO] Progress - 40% (7495/18736)
2015-02-06 12:51:21:532 [INFO] Progress - 50% (9368/18736)
2015-02-06 12:51:27:975 [INFO] Progress - 60% (11242/18736)
2015-02-06 12:51:36:860 [INFO] Progress - 70% (13116/18736)
2015-02-06 12:51:44:479 [INFO] Progress - 80% (14989/18736)
2015-02-06 12:51:54:500 [INFO] Progress - 90% (16863/18736)
2015-02-06 12:52:04:094 [INFO] Progress - 100% (18736/18736)
2015-02-06 12:52:04:095 [INFO] Analyzer - Pairwise - End
2015-02-06 12:52:04:095 [INFO] Analyzer - Write - Start
2015-02-06 12:52:04:905 [INFO] Analyzer - Write - End
2015-02-06 12:52:04:906 [INFO] Stopwatch - Total: 2:51s
2015-02-06 12:52:05:054 [INFO] Watcher - Prioritizing - Process completed
2015-02-06 12:52:05:054 [INFO] Watcher - New event - ID: 2563303858
2015-02-06 12:52:05:055 [INFO] Watcher - Database - Timestamp: 2015-02-06 12:45:25.000
2015-02-06 12:52:05:055 [INFO] Watcher - Database - Action: opened
2015-02-06 12:52:05:055 [INFO] Watcher - Database - Number: 880
2015-02-06 12:52:05:055 [INFO] Watcher - Database - Repository: gocd/gocd
2015-02-06 12:52:05:055 [WARN] Watcher - Skip - Repository directory does not exist
2015-02-06 12:52:05:055 [INFO] Watcher - New event - ID: 2563303838
2015-02-06 12:52:05:056 [INFO] Watcher - Database - Timestamp: 2015-02-06 12:45:24.000
2015-02-06 12:52:05:056 [INFO] Watcher - Database - Action: closed
2015-02-06 12:52:05:056 [INFO] Watcher - Database - Number: 2
```

```
2015-02-06 12:52:05:056 [INFO] Watcher - Database - Repository: BETaaS/BETaaS_Platform-Tools
2015-02-06 12:52:05:056 [WARN] Watcher - Skip - Repository directory does not exist
2015-02-06 12:52:05:056 [INFO] Watcher - New event - ID: 2563304073
2015-02-06 12:52:05:056 [INFO] Watcher - Database - Timestamp: 2015-02-06 12:45:32.000
2015-02-06 12:52:05:056 [INFO] Watcher - Database - Action: closed
2015-02-06 12:52:05:056 [INFO] Watcher - Database - Number: 2600
2015-02-06 12:52:05:056 [INFO] Watcher - Database - Repository: ReactiveX/RxJava
2015-02-06 12:52:05:057 [WARN] Watcher - Skip - Already up-to-date with respect to the event
...
2015-02-06 13:53:08:337 [INFO] Watcher - New event - ID: 2563420969
2015-02-06 13:53:08:338 [INFO] Watcher - Database - Timestamp: 2015-02-06 13:53:03.000
2015-02-06 13:53:08:338 [INFO] Watcher - Database - Action: opened
2015-02-06 13:53:08:338 [INFO] Watcher - Database - Number: 10615
2015-02-06 13:53:08:338 [INFO] Watcher - Database - Repository: docker/docker
2015-02-06 13:53:08:344 [INFO] Watcher - Prioritizing - Start process
2015-02-06 13:53:08:691 [INFO] Analyzer - Setup - Done
2015-02-06 13:53:08:761 [INFO] Analyzer - Fetch - Start
2015-02-06 13:53:09:221 [INFO] Trainer - Skip - Already recently updated
2015-02-06 13:53:54:345 [INFO] Analyzer - Fetch - End
2015-02-06 13:53:54:345 [INFO] Analyzer - Single - Start
2015-02-06 13:53:55:752 [INFO] Progress - 10% (14/140)
2015-02-06 13:53:58:323 [INFO] Progress - 20% (28/140)
2015-02-06 13:53:58:752 [INFO] Progress - 30% (42/140)
2015-02-06 13:53:59:039 [INFO] Progress - 40% (56/140)
2015-02-06 13:53:59:460 [INFO] Progress - 50% (70/140)
2015-02-06 13:53:59:846 [INFO] Progress - 60% (84/140)
2015-02-06 13:54:00:298 [INFO] Progress - 70% (98/140)
2015-02-06 13:54:00:660 [INFO] Progress - 80% (112/140)
2015-02-06 13:54:00:968 [INFO] Progress - 90% (126/140)
2015-02-06 13:54:01:381 [INFO] Progress - 100% (140/140)
2015-02-06 13:54:01:381 [INFO] Analyzer - Single - End
2015-02-06 13:54:01:382 [INFO] Analyzer - Total - Start
2015-02-06 13:54:01:771 [INFO] Predictor - Prediction - Start
2015-02-06 13:54:02:290 [INFO] Predictor - Prediction - End
2015-02-06 13:54:02:335 [INFO] Analyzer - Total - End
2015-02-06 13:54:02:336 [INFO] Analyzer - Pairwise - Start
2015-02-06 13:54:09:781 [INFO] Progress - 10% (960/9592)
2015-02-06 13:54:14:740 [INFO] Progress - 20% (1919/9592)
2015-02-06 13:54:22:263 [INFO] Progress - 30% (2878/9592)
2015-02-06 13:54:27:022 [INFO] Progress - 40% (3837/9592)
2015-02-06 13:54:30:467 [INFO] Progress - 50% (4796/9592)
2015-02-06 13:54:34:131 [INFO] Progress - 60% (5756/9592)
2015-02-06 13:54:38:790 [INFO] Progress - 70% (6715/9592)
2015-02-06 13:54:41:374 [INFO] Progress - 80% (7674/9592)
2015-02-06 13:54:43:919 [INFO] Progress - 90% (8633/9592)
2015-02-06 13:54:51:795 [INFO] Progress - 100% (9592/9592)
2015-02-06 13:54:51:795 [INFO] Analyzer - Pairwise - End
2015-02-06 13:54:51:795 [INFO] Analyzer - Write - Start
2015-02-06 13:54:52:545 [INFO] Analyzer - Write - End
2015-02-06 13:54:52:545 [INFO] Stopwatch - Total: 1:44s
2015-02-06 13:54:52:642 [INFO] Watcher - Prioritizing - Process completed
2015-02-06 13:54:52:642 [INFO] Watcher - New event - ID: 2563421134
2015-02-06 13:54:52:644 [INFO] Watcher - Database - Timestamp: 2015-02-06 13:53:08.000
2015-02-06 13:54:52:644 [INFO] Watcher - Database - Action: closed
2015-02-06 13:54:52:644 [INFO] Watcher - Database - Number: 88
2015-02-06 13:54:52:644 [INFO] Watcher - Database - Repository: anticoders/gagarin
2015-02-06 13:54:52:644 [WARN] Watcher - Skip - Repository directory does not exist
2015-02-06 13:54:52:644 [INFO] Watcher - New event - ID: 2563421464
2015-02-06 13:54:52:645 [INFO] Watcher - Database - Timestamp: 2015-02-06 13:53:20.000
2015-02-06 13:54:52:645 [INFO] Watcher - Database - Action: opened
2015-02-06 13:54:52:645 [INFO] Watcher - Database - Number: 26
2015-02-06 13:54:52:645 [INFO] Watcher - Database - Repository: Youssef-Emad/sQuiz
2015-02-06 13:54:52:645 [WARN] Watcher - Skip - Repository directory does not exist
2015-02-06 13:54:52:645 [INFO] Watcher - New event - ID: 2563421570
2015-02-06 13:54:52:646 [INFO] Watcher - Database - Timestamp: 2015-02-06 13:53:24.000
2015-02-06 13:54:52:646 [INFO] Watcher - Database - Action: closed
2015-02-06 13:54:52:646 [INFO] Watcher - Database - Number: 1
2015-02-06 13:54:52:646 [INFO] Watcher - Database - Repository: samsonos/php_upload
2015-02-06 13:54:52:646 [WARN] Watcher - Skip - Repository directory does not exist
```

Appendix C

Project list

01org\web-simulator
4teamwork\opengever.core
activiti\activiti
adaptivecomputing\torque
adlnet\adl_lrs
adobe\brackets
ajaxorg\cloud9
akeneo\pim-community-dev
akka\akka
alohaeditor\aloha-editor
amber-smalltalk\amber
andrewplummer\sugar
andris9\nodemailer
angband\angband
angular\angular.js
angular-ui\ng-grid
ansible\ansible
antirez\redis
antlr\antlr4
aodn\aodn-portal
apache\trafficserver
appium\appium
aptana\studio3
aptivate\econsensus
araq\nimrod
arcbees\arcbees-tools
arcbees\gwtp
arcbees\jukito
ariya\phantomjs
arokem\python-matlab-bridge
aspp\pelita
assaf\zombie
astropy\astropy
aterrien\jquery-knob
atom\atom
autotest\autotest
autotest\virt-test
ayende\ravendb
azure\azure-sdk-for-net
backlogs\redmine_backlogs
baystation12\baystation12
behat\mink
behat\minkselenium2driver
bem\bem-bl
bem\bem-components
bem\bem-core
biopython\biopython
bitcoin\bitcoin
bluemountaincapital\deedle
bnmnetp\runestone
boersmamarcel\challengesplatform
bokeh\bokeh
borisyankov\definitelytyped
buttonmen-dev\buttonmen
caelum\caelum-stella
caelum\vraptor
caelum\vraptor4
cakephp\cakephp
candlepin\candlepin
candlepin\subscription-manager
caolan\forms
cargomedia\puppet-packages
caskroom\homebrew-cask
ccnet\cruisecontrol.net
cdnjs\cdnjs
ceph\ceph
ceph\ceph-cookbook
ceph\teuthology
cesnet\perun
cfengine\core
cfengine\design-center
cfpb\regulations-parser
chiliproject\chiliproject
ci-bonfire\bonfire
citation-style-language\styles
cjohansen\sinon.js
clawpack\pyclaw
cleverraven\cataclysm-dda
cloudfoundry\cli
cloudname\cloudname
clusterlabs\pacemaker
cockpit-project\cockpit
cocoalumberjack\cocoalumberjack
codemontagehq\codemontage
commonjobs\commonjobs
composer\composer
connexions\cnx-archive
contiki-os\contiki
coreos\etcd
corsixth\corsixth
crawljax\crawljax
creationix\nvm
crosswalk-project\crosswalk
cvmfs\cvmfs
dam5s\happymapper
dana-i2cat\opennaas
dcache\dcache
deegree\deegree3
defnull\bottle
deis\deis

detro\ghostdriver
dgoodwin\tito
diaspora\diaspora
discourse\discourse
django\django
django-oscar\django-oscar
docker\docker
doctrine\phpcr-odm
droidplanner\droidplanner
drush-ops\drush
duckduckgo\community-platform
duckduckgo\duckduckgo
duckduckgo\zeroclickinfo-fathead
duckduckgo\zeroclickinfo-goodies
duckduckgo\zeroclickinfo-spice
edx\configuration
edx\edx-ora2
edx\edx-platform
einaros\ws
elasticsearch\elasticsearch
elasticsearch\elasticsearch-net
elasticsearch\logstash
elkarte\elkarte
emberjs\ember.js
enonic\wem-ce
enyojs\ares-project
ericdrowell\kineticjs
es-shims\es5-shim
escoz\quickdialog
ethersex\ethersex
etmc\tmlqcd
ets-berkeley-edu\calcentral
ettercap\ettercap
eucalyptus\eutester
exacttarget\fuelux
exercism\exercism.io
eyescale\equalizer
ezsystems\ezpublish-kernel
fabric\fabric
facebook\hhvm
facebook\react
fakeiteasy\fakeiteasy
fatfreecrm\fat_free_crm
fcrepo4\fcrepo4
feincms\feincms
feldspar\feldspar-compiler
findwise\hydra
firebug\firebug
firedrakeproject\firedrake
fluid-project\infusion
fog\fog
fontforge\fontforge
frappe\erpnext
freeradius\freeradius-server
freeseer\freeseer
frozennode\laravel-administrator
fuel\core
futuretap\inappsettingskit
galsim-developers\galsim
gamua\starling-framework
gazler\githug
geoadmin\mf-chsdi3
geonode\geonode
getnikola\nikola
github\choosealicense.com
github\linguist
google\traceur-compiler
groovy\groovy-core
growstuff\growstuff
guard\guard
h5bp\html5-boilerplate
haxeflixel\flixel
hazelcast\hazelcast
hollie\misterhouse
homebrew\homebrew
homebrew\homebrew-science
honestbleeps\reddit-enhancement-suite
hpcc-systems\hpcc-platform
hpcugent\easybuild-easyblocks
hpcugent\easybuild-easyconfigs
hpcugent\easybuild-framework
hrydgard\ppsspp
hylang\hy
hypothesis\h
icsharpcode\nrefactory
icsharpcode\sharpdevelop
idan\oauthlib
ilios\ilios
imaginationforpeople\imaginationforpeople
imathis\octopress
incubaid\arakoon
integrity\integrity
ipython\ipython
isagalaev\highlight.js
isensedev\rsense
jasig\cas
jbosstools\jbosstools-vpe
jch\html-pipeline
jejacks0n\mercury
jekyll\jekyll
jenkinsci\email-ext-plugin
jenkinsci\jenkins
jermolene\tiddlywiki5
jetbrains\intellij-community
jfinkels\flask-restless
jfryman\puppet-nginx
joindin\joind.in
joomla\joomla-cms
joyent\node
jquery\jquery
jquery\jquery-mobile
jscs-dev\node-jscs
jsdelivr\jsdelivr
juju\juju-gui
kaltura\mwembed
kanaka\novnc
kanaka\websockify
katello\katello
kennyledet\algorithm-implementations
keplerproject\luarocks
kliment\printrun
kogmbh\webodf
kohana\core
kotti\kotti
kripken\emscripten
kriskowal\q
learningregistry\learningregistry
ledger\ledger
less\less.js
libmesh\libmesh
libopencm3\libopencm3
librenms\librenms
libretro\retroarch
limesurvey\limesurvey
liqd\adhocracy
lobid\lodmill
loopj\android-async-http
lusitanian\phpoauthlib
macoslib\macoslib
maescool\catacomb-snatch
magit\magit
maglev\maglev
marionettejs\backbone.marionette
masdennis\rajawali
mbostock\d3
mc-server\mcserver
medinria\medinria-public
medo42\gang-garrison-2

mhevery\jasmine-node
midgetspy\sick-beard
minetest\minetest
mistio\mist.io
mne-tools\mne-python
mochi\mochiweb
mongodb\mongo
mongodb\mongo-java-driver
mongodb\mongo-php-driver
mongodb\node-mongodb-native
mopidy\mopidy
mozilla\bedrock
mozilla\fxa-content-server
mozilla\kitsune
mozilla\mozillians
mozilla\mozstumbler
mozilla\moztrap
mozilla\pdf.js
mozilla\remo
mozilla\socorro
mozilla\treeherder-service
mozilla\treeherder-ui
mozilla\webmaker.org
mozilla-appmaker\appmaker
mozilla-b2g\platform_hardware_ril
mozilla-services\cornice
mozilla-services\heka
mpc-hc\mpc-hc
mrdoob\three.js
mrjoes\flask-admin
mroth\lolcommits
mruby\mruby
mulesoft\mule
mumble-voip\mumble
munin-monitoring\munin
musicbrainz\picard
mysociety\alaveteli
mytardis\mytardis
nanoc\nanoc
naparuba\shinken
nengo\nengo
netty\netty
nipy\dipy
nla\banjo
node-inspector\node-inspector
nodejitsu\node-http-proxy
nojhan\liquidprompt
numpy\numpy
ocaml\opam
omab\django-social-auth
omab\python-social-auth
onaio\onadata
onepercentclub\onepercentclub-site
opencog\opencog
openframeworks\openframeworks
opengamma\og-platform
openlayers\openlayers
openmicroscopy\openmicroscopy
openscholar\openscholar
openseadragon\openseadragon
openslides\openslides
opm\opm-core
opscode\chef
opscode\chef-docs
opscode\knife-windows
opscode\omnibus
opscode\supermarket
orcid\orcid-source
osiam\server
osmandapp\osmand
otwcode\otwarchive
overviewer\minecraft-overviewer
owncloud\core
paparazzi\paparazzi
paulmillr\es6-shim
phalcon\cphalcon
phingofficial\phing
photo\mobile-android
photonstorm\phaser
phpdocumentor\phpdocumentor2
phpmyadmin\phpmyadmin
phpoffice\phpexcel
phunehehe\chef-cookbooks
pkp\ojs
playframework\playframework
plomino\plomino
powerdns\pdns
praekelt\vumi
praekelt\vumi-go
praekelt\vumi-jssandbox-toolkit
praw-dev\praw
presidentbeef\brakeman
project-osrm\osrm-backend
projecthydra\active_fedora
prose\prose
pulp\pulp
puma\puma
puppetlabs\puppet
puppetlabs\puppetlabs-training-bootstrap
pyinstaller\pyinstaller
pyne\pyne
python-pillow\pillow
qreal\qreal
qtile\qtile
quattor\configuration-modules-core
quicksilver\quicksilver
qupzilla\qupzilla
rack\rack
rackspace\php-opencloud
rbschange\change
reactiveui\reactiveui
reactivex\rxjava
readytalk\avian
refinery\refinerycms
rejectedsoftware\vibe.d
repoforge\rpms
request\request
revel\revel
rgbkrk\atom-script
riot-os\riot
robbiehanson\cocoaasyncsocket
robbiehanson\xmppframework
robbyrussell\oh-my-zsh
ros\rosdistro
ros-visualization\rviz
rosedu\wouso
roundcube\roundcubemail
ruflin\elastica
rust-lang\rust
sage-bionetworks\synapse-repository-services
sagemath\sagenb
saltstack\salt
sapo\ink
sbadia\puppet-gitlab
sc3\cookcountyjail
scala-js\scala-js
schambers\fluentmigrator
scikit-image\scikit-image
scikit-learn\scikit-learn
scitools\iris
scummvm\scummvm
segmentio\analytics-ios
sehmaschine\django-grappelli
seomoz\qless
servo\servo
sharekit\sharekit
shellycloud\shelly
shish\shimmie2

shogun-toolbox\shogun
shower\shower
simbul\baker
sinatra\sinatra
sjkaliski\numbers.js
skyscreamer\jsonassert
skyscreamer\yoga
slick\slick
snoyberg\conduit
socketstream\socketstream
sonata-project\sonataadminbundle
sonatype\nexus-oss
sosreport\sos
spiral-project\daybed
splitbrain\dokuwiki
spray\spray
sproutcore\sproutcore
stan-dev\stan
stanford-online\class2go
stepcode\stepcode
substack\node-browserify
sunpy\sunpy
swanson\stringer
symfony\symfony
symfony-fr\symfony-docs-fr
symphonycms\symphony-2
sympy\sympy
tastejs\todomvc
taulabs\taulabs
technomancy\leiningen
telmich\cdist
ten24\slatwall
texttochange\vusion-backend
texttochange\vusion-frontend
theforeman\foreman
thoughtbot\factory_girl
tinymce\tinymce
tombatossals\angular-leaflet-directive
translate\pootle
travis-ci\travis-core
twall\jna
twbs\bootstrap
twitter\commons
ufz\ogs
unepwcmc\sapi
unknown-horizons\unknown-horizons
venomous0x\whatsapi
videojs\video.js
void256\nifty-gui
voldemort\voldemort
vufind-org\vufind
w3c\web-platform-tests
waterbearlang\waterbear
wet-boew\wet-boew
wikia\app
wikimedia\jquery.uls
windower\lua
wiredtiger\wiredtiger
woothemes\woocommerce
wordpress-mobile\wordpress-ios
wp-cli\wp-cli
xapi-project\xen-api
xbmc\xbmc
xdebug\xdebug
xp-framework\xp-framework
xwiki\xwiki-platform
yandex-ui\nanoislands
yeoman\generator
yesodweb\persistent
yesodweb\wai
yesodweb\yesod
yiisoft\yii2
yusuke\twitter4j
zanata\zanata-server
zendframework\zf2
zeromq\jzmq
zfsonlinux\spl
zfsonlinux\zfs
znc\znc

Appendix D

Queries

```
select u.login, p.name, (
  select count(*) as total_created
  from pull_requests pr, pull_request_history prh
  where p.id = pr.base_repo_id
  and prh.action = 'opened'
  and prh.pull_request_id = pr.id
  and prh.created_at > DATE_SUB(now(), INTERVAL 6 MONTH)
) / 180 as avg_prs_per_day
from projects p, users u
where u.id = p.owner_id
  and p.deleted is false
```

Listing D.1: Gets the average number of pull requests per day from GHTorrent.

```sql
select u.login, p.name, (
  (
    select count(*) as total_created
    from pull_requests pr, pull_request_history prh
    where p.id = pr.base_repo_id
    and prh.pull_request_id = pr.id
    and prh.created_at > DATE_SUB(now(), INTERVAL 6 MONTH)
  ) + (
    select count(*) as total_created
    from pull_requests pr, pull_request_comments prc
    where p.id = pr.base_repo_id
    and prc.pull_request_id = pr.id
    and prc.created_at > DATE_SUB(now(), INTERVAL 6 MONTH)
  ) + (
    select count(*) as total_created
    from issues i, pull_requests pr, issue_events ie
    where p.id = i.repo_id
    and i.pull_request_id = pr.id
    and ie.issue_id = i.id
    and ie.created_at > DATE_SUB(now(), INTERVAL 6 MONTH)
  ) + (
    select count(*) as total_created
    from issues i, pull_requests pr, issue_comments ic
    where p.id = i.repo_id
    and i.pull_request_id = pr.id
    and ic.issue_id = i.id
    and ic.created_at > DATE_SUB(now(), INTERVAL 6 MONTH)
  )
) / 180 as avg_actions_per_day
from projects p, users u
where u.id = p.owner_id
  and p.deleted is false
```

Listing D.2: Gets the average number of pull requests actions per day from GHTorrent.

Appendix E

Evaluation survey

Prioritizing pull requests

The past few months we performed a study to prioritization of pull requests.
As a result an (initial) prioritization service was developed.
You've been invited to test the service on your repository.
The PRioritizer service can be found at: http://ghtorrent.org/prioritizer/

After you've had the time to test the PRioritizer service, we would like to ask you to fill in this short questionnaire.

Thank you!

* Required

Which repository or repositories do you handle pull requests for? *

E.g. github/octicons

The PRioritizer service

What do you like the most about the PRioritizer service?

Please rate the usefulness of the following features *

	Very useful	Useful	Not so useful	Useless
Automatic ordering	○	○	○	○
Test code filtering	○	○	○	○
Target branch filtering	○	○	○	○
Pairwise conflicts inspection/sorting	○	○	○	○
Contributed commits inspection/sorting	○	○	○	○
Accept rate inspection/sorting	○	○	○	○
Size inspection/sorting	○	○	○	○

What features do you miss from the PRioritizer?

Please rate the usability of the PRioritizer service *

	Strongly agree	Agree	Disagree	Strongly disagree
It is easy to get an overview of the state of the pull requests of the project	○	○	○	○
It is easy to find what pull request to work on next	○	○	○	○
The filter lacks support for some important fields	○	○	○	○
Using the prioritizer service causes too much overhead	○	○	○	○
Some pull requests show incorrect information	○	○	○	○

Please rate the following aspects of the PRioritizer service *

	Strongly agree	Agree	Disagree	Strongly disagree
I would use prioritizer for my project	○	○	○	○
I would recommend it to others	○	○	○	○
I like the prioritizer web interface more than GitHub's	○	○	○	○
The performance of prioritizer is adequate	○	○	○	○
The time delay between GitHub and the service is acceptable	○	○	○	○

Please rate the following features you would like to see in a future version

	Very useful	Useful	Not so useful	Useless
Prioritization per integrator instead of per repository (triage)	○	○	○	○
Support for private repositories	○	○	○	○
Support for GitHub labels	○	○	○	○
Filter on pull request type (e.g. fix/refactor/feature/doc)	○	○	○	○
I would like to have more control over the prioritization (user feedback)	○	○	○	○

How can we improve the PRioritizer service for your project?

Would you be available for a short interview over email, Skype or Google Hangouts?

If so, please fill in your email address