Tampere University

Vili Nikkola

CODE QUALITY IN PULL REQUESTS, AN EMPIRICAL STUDY

Faculty of Information Technology and Communication Sciences
Master of Science Thesis
August 2019

ABSTRACT

Vili Nikkola: Code quality in pull requests, an empirical study
Master of Science Thesis
Tampere University
Information Technology
August 2019

Pull requests are a common practice for contributing and reviewing contributions, and are employed both in open-source and industrial contexts. Compared to the traditional code review process adopted in the 1970s and 1980s, pull requests allow a more lightweight reviewing approach. One of the main goals of code reviews is to find defects in the code, allowing project maintainers to easily integrate external contributions into a project and discuss the code contributions.

The goal of this work is to understand whether code quality is actually considered when pull requests are accepted. Specifically, we aim at understanding whether code quality issues such as code smells, antipatterns, and coding style violations in the pull request code affect the chance of its acceptance when reviewed by a maintainer of the project.

We conducted a case study among 28 Java open-source projects, analyzing the presence of 4.7 M code quality issues in 36 K pull requests. We analyzed further correlations by applying Logistic Regression and seven machine learning techniques (Decision Tree, Random Forest, Extremely Randomized Trees, AdaBoost, Gradient Boosting, XGBoost).

Unexpectedly, code quality turned out not to affect the acceptance of a pull request at all. As suggested by other works, other factors such as the reputation of the maintainer and the importance of the feature delivered might be more important than code quality in terms of pull request acceptance.

Keywords: Pull Request, Software Quality, Empirical Study

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

TIIVISTELMÄ

Vili Nikkola: Koodin laadukkuuden vaikutus pull requesteihin, empiirinen tutkimus
Diplomityö
Tampereen yliopisto
Tietotekniikka
Elokuu 2019

Pull requestit ovat yleinen mekanismi osallistua sovelluskehitykseen avoimen lähdekoodin projekteissa sekä työmaailmassa. Verrattuna perinteisempään koodin katselmointiprosessiin, pull requestit tarjoavat kevyemmän vaihtoehdon suorittaa koodikatselmus. Koodikatselmoinnin yksi tärkeimmistä tehtävistä on löytää lähdekoodista virheitä, joiden löytyminen helpoittaa uuden lähdekoodin integrointia projektiin.

Tämän työn tavoitteena on ymmärtää pull requestien laatutekijöiden vaikutus sen hyväksyntään ja sitä kautta integrointiin projektiin. Työssä keskitytään erityisesti koodista löytyviin ongelmiin kuten lähdekoodin suunnitteluvirheisiin, rakennevirheisiin ja tyylivirheisiin ja niiden vaikutukseen hyväksymistuloksessa.

Työssä tutkittiin 28:aa avoimen lähdekoodin Java-projektia ja analysoitiin noin 4,7 miljoonaa lähdekoodiriviä yhteensä 36 tuhannessa pull requestissa. Laatuongelmien korrelaatioita analysoitiin käyttäen hyväksi logistista regressiota sekä seitsemää eri koneoppimisen menetelmää (Decision trees, Random Forest, Extremely Randomized Trees, AdaBoost, Gradient Boosting, XGBoost).

Työn tekijän yllätykseksi laatuongelmien esiintyminen pull requestien lähdekoodissa ei näytä vaikuttavan sen hyväksyntään ja sitä kautta sisällyttämiseen kohdeprojektin lähdekoodiin. Aikaisempi tutkimus osoittaa, muut tekijät kuten pull requestin luojan maine ohjelmoijana sekä lähdekoodilisäyksen tuottaman lisäominaisuuden tärkeys saattavat olla tärkeämpiä tekijöitä pull requestien hyväksymisprosessissa.

Avainsanat: Pull Request, Ohjelmistojen laatu, Empiirinen tutkimus

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

PREFACE

This Master of Science Thesis was made for the department of Information Technology and Communications Sciences of Tampere University (TUNI). The thesis was made under the supervision of Davide Taibi and Valentina Lenarduzzi.

I would like to thank Davide Taibi and Valentina Lenarduzzi for their invaluable help in formulating the idea for this study and guiding me throughout the research and writing process. I also want to thank Nyyti Saarimäki for her help with the study.

Special thanks to the Guild of Information Technology for making my time at the university so enjoyable.

Tampere, 4th August 2019

Vili Nikkola

CONTENTS

1 Introduction 1
  1.1 Introduction 1
2 Background 3
  2.1 Measuring code quality 3
    2.1.1 Software program analysis 3
    2.1.2 PMD 4
  2.2 Version Control 4
    2.2.1 Distributed Version Control 6
    2.2.2 Git 6
    2.2.3 GitHub 7
    2.2.4 Pull Requests 7
  2.3 Machine Learning 8
    2.3.1 Machine Learning Scenarios 9
    2.3.2 Machine Learning Techniques 11
3 Related Work 15
  3.1 Related Work 15
    3.1.1 Pull Request Process 15
    3.1.2 Software Quality of Pull Requests 16
4 Case Study Design 18
  4.1 Case Study Design 18
    4.1.1 Goal and Research Questions 18
    4.1.2 Context 19
    4.1.3 Data Collection 21
    4.1.4 Data Analysis 21
    4.1.5 Replicability 23
5 Implementation 24
  5.1 Software structure and design decisions 24
    5.1.1 Docker 24
    5.1.2 Node.js 25
    5.1.3 Typescript 25
    5.1.4 Machine Learning with Python 25
  5.2 Software structure and considerations 26
6 Results 31
7 Discussion 38
8 Threats to validity 39
9 Conclusion 41

References 42

LIST OF FIGURES

2.1 Git branching and pull requests visualized 8
2.2 Example supervised learning input data 9
2.3 Example of a clustering algorithm result 10

5.1 The structure of the software implemented 28

6.1 ROC Curves of Adaboost and Bagging - (RQ2) 34
6.2 ROC Curves of DecitionTrees and ExtraTrees - (RQ2) 36
6.3 ROC Curves of GradientBoost and LogisticRegression - (RQ2) 36
6.4 ROC Curves of RandomForest and XGBoost - (RQ2) 36

LIST OF TABLES

2.1 Example of PMD rules and their related harmfulness 5

4.1 Selected projects 20
4.2 Example of data structure used for the analysis 21
4.3 Accuracy measures 22

6.2 Distribution of TD items in pull requests - (RQ1) 31
6.1 Distribution of pull requests (PR) and technical debt items (TD items) in the selected projects - (RQ1) 32
6.3 Descriptive statistics (the 20 most recurrent TD items) - (RQ1) 33
6.4 Model reliability - (RQ2) 34
6.5 Summary of the quality rules related to pull request acceptance - (RQ2 and RQ3) 35
6.6 Contingency matrix 35

LIST OF PROGRAMS AND ALGORITHMS

5.1 Docker container definition 26
5.2 Example of the main function 29
5.3 Example of PMD analysis 30

LIST OF SYMBOLS AND ABBREVIATIONS

AUC	Area Under The Receiver Operating Characteristic
CSV	Comma-separated values
CVCS	Centralized Version Control System
MCC	Matthew Correlation Coefficient
Pull Request	A unit of contribution to project in a distributed version control system
ROC	Receiver Operating Characteristics
TD	Technical Debts
VCS	Version Control System

1 INTRODUCTION

1.1 Introduction

Different code review techniques have been proposed in the past and widely adopted by open-source and commercial projects. Code reviews involve the manual inspection of the code by different developers and help companies to reduce the number of defects and improve the quality of software [2][1].

Nowadays, code reviews are generally no longer conducted as they were in the past, when developers organized review meetings to inspect the code line by line [24].

The industry and researchers are in agreement today that code inspection helps to reduce the number of defects, but that in some cases, the effort required to perform code inspections hinders their adoption in practice [63]. However, the boon of new tools and practices has enabled companies to adopt more lightweight code review practices. In particular, several companies, including Facebook [25], Google [53], and Microsoft [6], perform code reviews by means of tools such as Gerrit[1] or the pull request mechanism provided by Git[2] [58].

In the context of this thesis, we focus on pull requests. Pull requests provide developers a convenient way of contributing to projects, and many popular projects, including both open-source and commercial ones, are using pull requests as a way of reviewing the contributions of different developers.

Researchers have focused their attention on pull request mechanisms, investigating different aspects, including the review process [32], [31] and [68], the influence of code reviews on continuous integration builds [74], how pull requests are assigned to different reviewers [73], and in which conditions they are accepted process [32],[56],[65],[39].

Only a few works have investigated whether developers consider quality aspects in order to accept pull requests [32],[31].

Different works report that the reputation of the developer who submitted the pull request is one of the most important acceptance factors [31],[16].

However, to the best of our knowledge, no studies have investigated whether the quality of the code submitted in a pull request has an impact on the acceptance of this pull

[1] https://www.gerritcodereview.com
[2] https://help.github.com/en/articles/about-pull-requests

request. As code reviews are a fundamental aspect of pull requests, we strongly expect that pull requests containing low-quality code should generally not be accepted.

In order to understand whether code quality is one of the acceptance drivers of pull requests, we designed and conducted a case study involving 28 well-known Java projects to analyze the quality of more than 36K pull requests. We analyzed the quality of pull requests using PMD[3], one of the four tools used most frequently for software analysis [43], [8]. PMD evaluates the code quality against a standard rule set available for the major languages, allowing the detection of different quality aspects generally considered harmful, including code smells [27] such as "long methods", "large class", "duplicated code"; anti-patterns [15] such as "high coupling"; design issues such as "god class" [40]; and various coding style violations[4]. Whenever a rule is violated, PMD raises an issue that is counted as part of the Technical Debt [20]. In the remainder of this work, we will refer to all the issues raised by PMD as "TD items" (Technical Debt items).

Previous work confirmed that the presence of several code smells and anti-patterns, including those collected by PMD, significantly increases the risk of faults on the one hand and maintenance effort on the other hand [37], [50], [21], [26].

Unexpectedly, our results show that the presence of TD items of all types does not influence the acceptance or rejection of a pull request at all. To prove this statement, we analyzed all the data not only using basic statistical techniques, but also applying seven machine learning algorithms (Logistic Regression, Decision Tree, Random Forest, Extremely Randomized Trees, AdaBoost, Gradient Boosting, XGBoost), analyzing 36,986 pull requests and over 4.6 million TD items present in the pull requests.

Structure of the thesis. Section 2 describes the basic concepts underlying this work, while Section 3 presents some related work done by researchers in recent years. In Section 4, we describe the design of our case study, defining the research questions, metrics, and hypotheses, and describing the study context, including the data collection and data analysis protocol. In section 5 the implementation details of the software used are presented. In Section 6, we present the achieved results and discuss them in Section 7. Section 8 identifies the threats to the validity of our study, and in Section 9, we draw conclusions from the information gathered in the study.

[3]https://pmd.github.io
[4]https://pmd.github.io/latest/pmd_rules_java.html

2 BACKGROUND

In this Section, we will first introduce code quality aspects and PMD, the tool we used to analyze the code quality of the pull requests. Then we will describe the pull request mechanism and finally provide a brief introduction and motivation for the usage of the machine learning techniques we applied.

2.1 Measuring code quality

Code quality is a loose approximation of the long-term usefulness and the long-term maintainability of code. ISO25010 identifies 8 main aspects of software quality in relation to quality evaluation: *Functional Suitability, Performance efficiency, Compatibility, Usability, Reliability, Security, Maintainability, Portability* [34]. While measuring functional suitability and usability would be hard to measure just be examining the source of a chosen project, other aspects of software quality can be measured through the use of source code analysis. Code analysis software can find patterns or behaviour in the source code that is associated with an increased risk of issues i.e. in the maintainability of the project.

2.1.1 Software program analysis

In order to gain an insight into the quality of the software program, a suitable approach to analysis must be decided. Software program analysis can be broadly divided into two categories: static code analysis for when the program is analyzed without executing it and dynamic program analysis for when the program's behaviour is monitored during execution. The analyzer can also employ a combination of these to categories.

As we are interested in analyzing pull requests of Java projects and dynamic program analysis requires the program to be compiled and executed, we can rule out the possibility of using dynamic analysis to measure the the quality of the source code. As pull requests are proposed changes to the existing project rather than a whole program and the compilation process for each project might have variations, we decided to use a static analyzer to gain knowledge about source code added or modified in the pull requests. Static analysis of have been found to be effective and complementary to dynamic software analysis via testing the program during its execution[69].

2.1.2 PMD

Different tools on the market can be used to evaluate code quality through program analysis. PMD is one of the most frequently used static code analysis tools for Java on the market, along with Checkstyle, Findbugs, and SonarQube [43]. All these tools aim to analyze the source code and provide users with insights into the quality of their code.

PMD is an open-source tool that aims to identify issues that can lead to technical debt accumulating during development. The specified source files are analyzed and the code is checked with the help of predefined rule sets. PMD provides a standard rule set for major languages, which the user can customize if needed. The default Java rule set encompasses all available Java rules in the PMD project and is used throughout this study.

Issues found by PMD have five priority values (P). Rule priority guidelines for default and custom-made rules can be found in the PMD project documentation [4].

- P1 Change absolutely required. Behavior is critically broken/buggy.
- P2 Change highly recommended. Behavior is quite likely to be broken/buggy.
- P3 Change recommended. Behavior is confusing, perhaps buggy, and/or against standards/best practices.
- P4 Change optional. Behavior is not likely to be buggy, but more just flies in the face of standards/style/good taste.
- P5 Change highly optional. Nice to have, such as a consistent naming policy for package/class/fields...

These priorities are used in this study to help determine whether more severe issues affect the rate of acceptance in pull requests.

PMD is a static code analyzer which does not require compiling the code to be analyzed. As the aim of our work was to analyze only the code of pull requests instead of the whole project code, we decided to adopt it. PMD defines more than 300 rules for Java, classified in eight categories (coding style, design, error prone, documentation, multithreading, performance, security). Several rules have also been confirmed harmful by different empirical studies. In Table 2.1 we highlight a subset of rules and the related empirical studies that confirmed their harmfulness. The complete set of rules is available on the PMD official documentation[4].

2.2 Version Control

Version control system (VCS) tracks changes to data in computer systems, in the context of this study the source code of the projects. VCS installed on to the user's computer can be used to create, organize and switch between the changed states of the codebase,

Table 2.1. *Example of PMD rules and their related harmfulness*

PMD Rule	Defined By	Impacted Characteristic
Avoid Using Hard-Coded IP	Brown et al [14]	Maintainability [14]
Loose Coupling	Chidamber and Kemerer [18]	Maintainability [3]
Base Class Should be Abstract	Brown et al [14]	Maintainability [37]
Coupling Between Objects	Chidamber and Kemerer [18]	Maintainability [3]
Cyclomatic Complexity	Mc Cabe [45]	Maintainability [3]
Data Class	Fowler [27]	Maintainability [44], Faultiness [64][70]
Excessive Class Length	Fowler (Large Class) [27]	Change Proneness [51][38]
Excessive Method Length	Fowler (Large Method) [27]	Change Proneness [35][38] Fault Proneness[51]
Excessive Parameter List	Fowler (Long Parameter List) [27]	Change Proneness [35]
God Class	Marinescu and Lanza [40]	Change Proneness [49][62][76], Comprehensibility [23], Faultiness [49][76]
Law of Demeter	Fowler (Inappropriate Intimacy) [27]	Change Proneness [51]
Loose Package Coupling	Chidamber and Kemerer [18]	Maintainability [3]
Comment Size	Fowler (Comments) [27]	Faultiness [5][4]

often called "revisions". These revisions can be further organized into branches that divert from the main timeline of the repository enabling the developer to work on multiple different versions of the same project at the same time and merge the different timelines back to the main timeline. The general process of branching and merging is illustrated in Figure 2.1. Revisions can be stored locally on the user's computer but this poses a problem when there are multiple developers working on the same project. To distribute the source code and keep all the developers' revisions available to others, client-server model can be utilized in the form of a centralized version control system (CVCS).

With CVCS the developer's version control system acts as the client and connects to a central code repository that provides the wanted revisions of the source code for the client to download. This centralized code repository provided a way for developers to commit their contributions to the project easily and access other developers' work at will. The main problem with a CVCS structure is that the entire code repository only exists in the designated code server, which has a risk of outages and failures. A failure in the CVCS server could halt development entirely or in the worst case scenario, the whole repository could be lost due to a failed hard drive. A centralized version control system also requires a connection to the server if a wants to commit a changeset to the

repository. The requirement of connecting to the server each time a commit is made can make remote and distributed development significantly harder.

As project became bigger and software development as a whole became increasingly distributed and centralized version control systems began showing these problems, many notable open source projects started switching to distributed version control systems[22] and distributed version controls have become the norm in the industry going forward.

2.2.1 Distributed Version Control

Distributed Version Control tries to alleviate the problems described before by having the source of the project available for download on the server, but instead of a single revision the user is given a local copy of the whole repository with all the known revisions to date. Having the whole repository distributed to all developers inherently provides resistance to data loss due to hardware problems or malicious tampering of the repository server. A local repository also enables the developer to commit a changeset into the repository and change branches of code without needing to connect to the server, allowing easier code management and faster operation because existing branches are fetched from the hard drive rather than downloaded from a remote source. Although many distributed version control systems exists today, in this study we focused on Git, which was found to be a popular choice among open source projects.

2.2.2 Git

Git[1] is a distributed version control system that enables users to collaborate on a coding project by offering a robust set of features to track changes to the code. Git was initially developed as a replacement to BitKeeper, a distributed version control system used in the development of the Linux kernel project since 2002. In 2005, the relationship between the Linux developer community and BitKeeper had turned sour and Torvalds with the whole Linux development community started the work on their own version control software that used the lessons learned from the time spent using BitKeeper[2].

Features include “committing” a change to a local repository, “pushing” that piece of code to a remote server for others to see and use, “pulling” other developers’ change sets onto the user’s workstation, and merging the changes into their own version of the code base. Changes can be organized into branches, which are used in conjunction with pull requests. Git provides the user a "diff" between two branches, which compares the branches and provides an easy method to analyze what kind of additions the pull request will bring to the project if accepted and merged into the master branch of the project.

[1] https://git-scm.com/

[2] https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git

With advanced features, continued development and the lack of licensing fees, Git has become the most preferred version control software among developers in the developer survey conducted by Stack Overflow by a large margin[3].

2.2.3 GitHub

With the popularity of Git increasing, many companies such as GitHub[4], Atlassian[5] and GitLab[6] started providing developers with remote version control services using Git. These service providers would handle the setup and maintenance of the Git server and also provide clients with additional services such as automation, communication and project management tools. These are among the reasons why Apache Software Foundation, the largest open source foundation made the decision to migrate their code projects to GitHub in early 2019[7]. Aside from Apache Software Foundation, GitHub provides code hosting to many notable entities, such as Google[8] and Microsoft[9], making it a good source of open source projects for this study.

2.2.4 Pull Requests

Pull requests are a code reviewing mechanism that is compatible with Git and are provided by GitHub. The goal is for code changes to be reviewed before they are inserted into the mainline branch. A developer can take these changes and push them to a remote repository on GitHub. Before merging or rebasing a new feature in, project maintainers in GitHub can review, accept, or reject a change based on the diff of the “master” code branch and the branch of the incoming change. Reviewers can comment and vote on the change in the GitHub web user interface. If the pull request is approved, it can be included in the master branch. A rejected pull request can be abandoned by closing it or the creator can further refine it based on the comments given and submit it again for review. A pull request can contain commits from the same repository as the target branch, but the pull request submitter could also have cloned the repository and stored their work on their own GitHub account before submitting the final result. This is also illustrated in Figure 2.1. The possibility of sharing work between repositories easily with pull requests provides developers more tools to organize their work and contribute to open source projects.

[3] https://insights.stackoverflow.com/survey/2018#work-_-version-control
[4] https://github.com/
[5] https://bitbucket.org/product
[6] https://about.gitlab.com/
[7] https://blogs.apache.org/foundation/entry/the-apache-software-foundation-expands
[8] https://github.com/google
[9] https://github.com/microsoft

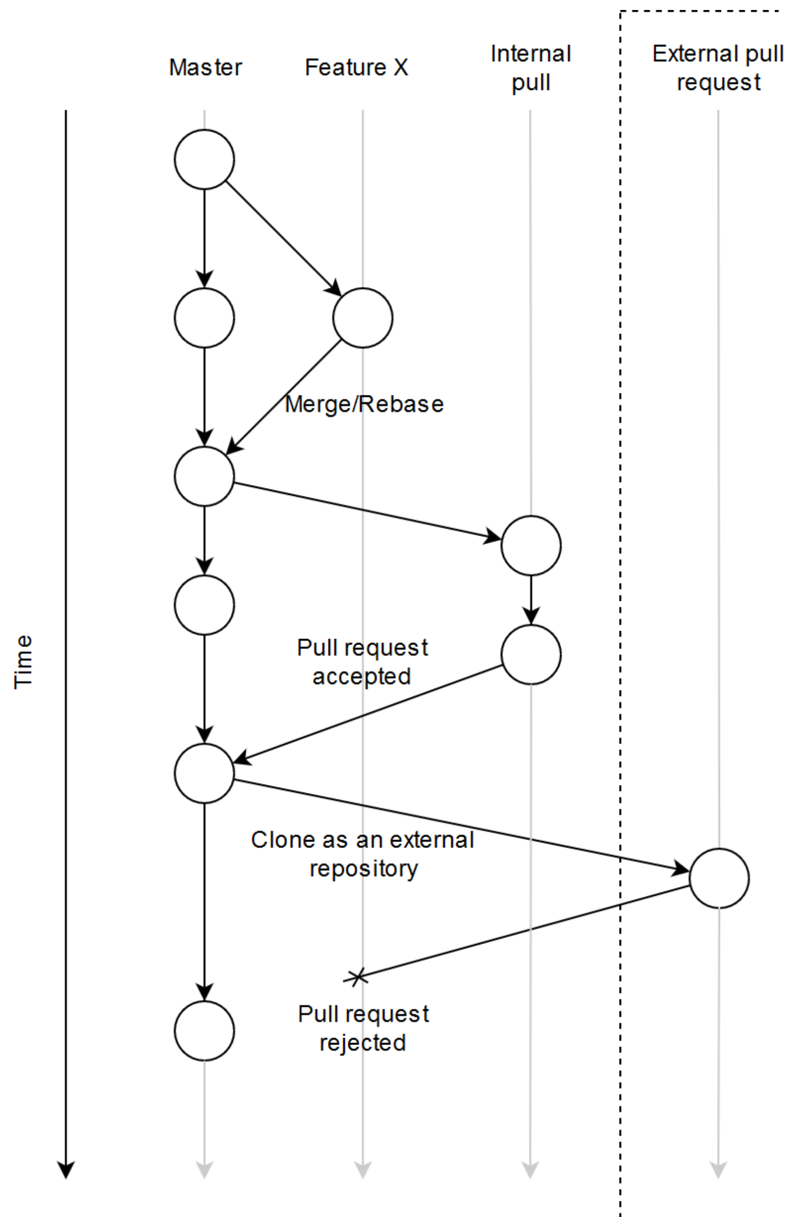


Figure 2.1. *Git branching and pull requests visualized*

2.3 Machine Learning

Measuring code quality in the code found in pull requests and the pull requests' acceptance generates a large dataset. In order to transform this dataset into something that can reveal something about about the relation of code quality and pull request acceptance, machine learning could be utilized. Tom M. Mitchell defines machine learning algorithms with: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E"[46]. In other words, a software that follows a some kind machine learning algorithm would be given a task that is evaluated based on some performance measurement and based on that measurement, the software can modify the execution of the task to achieve better performance. Tasks in machine learning can be

grouped into several general categories. In Foundations of Machine Learning, authors list 7 different common scenarios in machine learning with some common problems related to a scenario[47].

2.3.1 Machine Learning Scenarios

Supervised Learning is defined as "The learner receives a set of labeled examples as training data and makes predictions for all unseen points. This is the most common scenario associated with classification, regression, and ranking problems." In supervised learning, the software is given a pairs of data consisting a sample of the input that the algorithm will receive along with the observed outcome for those specific inputs. These pairs of data make up the training portion of the dataset which will result in a function, that can take in unseen data and provide the user with an output based on the learning that has happened in training the function.

Classifiers try to help the user to put a specific known label to a new data point. For example, a classifier can receive an image and estimate that the picture depicts a dog because it is similar to pictures in the training data that are labeled as dogs. Regression algorithms tries to give a numerical value to the data presented. It could be used to come up with a model that gives a price estimate on a product when comparing it to other similar products. A ranking algorithm can be used to fit a predetermined list of ranks to a group of things. For example, this years batch of produce could be automatically ranked based on the looks, weight and the color of the item. Figure 2.2 shows an example of labeled input data, where the dependant variable "Has heart disease" could be predicted with the help of independent variables, in this case other questions about the health of the patient.

Has heart disease	Overweight	High blood pressure	Chest pain	Resting heart rate > 100 bpm	...
yes	yes	yes	yes	yes	...
no	no	yes	yes	no	...
yes	no	yes	yes	yes	...
no	yes	no	no	no	...

Figure 2.2. *Example supervised learning input data*

Unsupervised Learning differs from supervised learning in that the training data isn't labeled. Without the explicit help of labels in the data, the algorithm performing unsupervised learning can try to find relationships between datapoints to group them in a meaningful way. This is referred to as clustering and is used for many practical purposes. Depicted in Figure 2.3 is an example of a clustered dataset. For example, image data can be fed into an image recognition algorithm which can use clustering to group the unseen image data with similar previously seen data, thus giving its own estimation on what the

image actually represents.

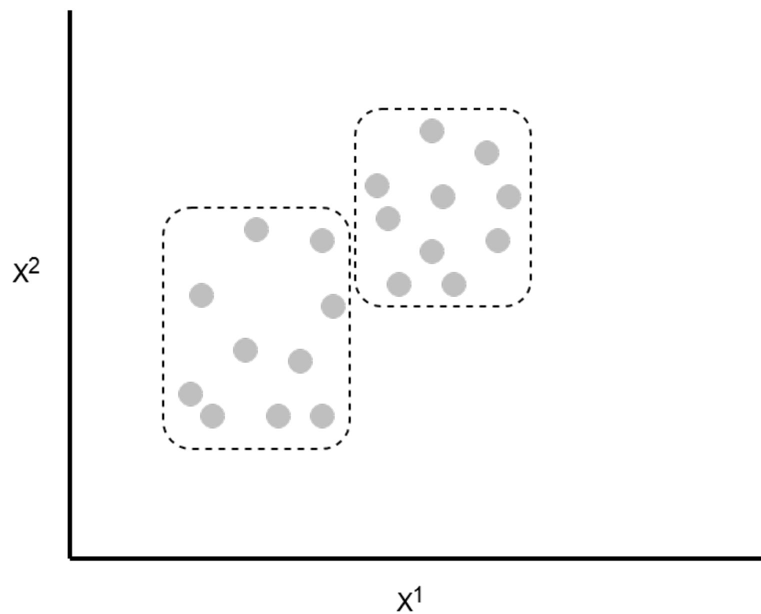


Figure 2.3. *Example of a clustering algorithm result*

In *Semi-supervised Learning* model receives training data consisting of both labeled and unlabeled data, and makes predictions for all new unseen datapoints [47]. This approach can help in situations, where collecting labeled data is expensive, such as in medical research. *Transductive Inference* is similar to semi-supervised learning, but rather than predicting labels for unseen data, transductive inference tries to label all the unlabeled datapoints in the training data.

In *On-line learning* the training is done in multiple instances, rather than with the whole dataset in one operation. After each instance of training, the model is tested with an unlabeled datapoint and the predictor is adjusted depending on the correctness of the prediction. On-line learning can be helpful when the dataset is too large to be trained with in one operation or using all of the data is not feasible. In *Active learning*, the learner adaptively or interactively collects training examples, typically by quering an oracle to request labels for new points. The goal in active learning is to achieve a performance comparable to standard supervised learning scenario, but with fewer labeled examples [47].

Reinforcement Learning also mixes training and testing phases but instead of giving the model a ready-made training set, the model is given an environment that it can affect with actions and those actions can lead to a reward. The aim of the training is to accumulate knowledge of what actions are needed to increase the chances of getting a reward. The challenge of this type of an approach to learning is the explorations versus exploitation dilemma, where the model must choose between exploring unknown actions to gain more information versus exploiting the information already collected. As an example of reinforcement learning, a computer program could be taught to play a game by giving it the possibility of playing the game through actions and providing the program feedback

for those actions taken in the form of points or other metrics. The program would then experiment with actions that are within the rules of the game and try to collect as many points as possible and effectively learning to play the game.

As both the input (code quality of the pull request) and the output (pull request acceptance) for a pull request can be observed, machine learning classifiers that use the principles of supervised learning can be used to analyzed the correlations between code quality and pull request acceptance.

2.3.2 Machine Learning Techniques

In this section, we will describe the machine learning classifiers adopted in this work. We used eight different classifiers: a generalized linear model (Logistic Regression), a tree-based classifier (Decision Tree), and six ensemble classifiers (Bagging, Random Forest, ExtraTrees, AdaBoost, GradientBoost, and XGBoost).

Ensemble type classifiers use a collection of simpler models to improve performance or solve problems that the models forming the collection might have. Classifiers using boosting [61] use a collection of weaker classifiers to create a stronger classifier. These weaker classifiers are created in sequence and modified or "grown" based on the results of the previous round of boosting.

In the next sub-sections, we will briefly introduce the eight adopted classifiers and give the rationale for choosing them for this study.

Logistic Regression

Logistic Regression [19] is one of the most frequently used algorithms in Machine Learning. In logistic regression, a collection of measurements (the counts of a particular issue) and their binary classification (pull request acceptance) can be turned into a function that outputs the probability of an input being classified as 1, or in our case, the probability of it being accepted.

To calculate the probability, logistic regression uses the sigmoid function, which is defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The coefficients for this function are calculated using maximum likelihood estimation, which aims to maximize the probability of finding the input measurements with the function. Logistic regression was chosen for this study because its wide adoption and ease of implementation.

DecisionTree

Decision Tree [10] is a model that takes learning data and constructs a tree-like graph of decisions that can be used to classify new input. The learning data is split into subsets based on how the split from the chosen variable improves the accuracy of the tree at the time. The decisions connecting the subsets of data form a flowchart-like structure that the model can use to tell the user how it would classify the input and how certain the prediction is perceived to be.

We considered two methods for determining how to split the learning data: GINI impurity and information gain. GINI tells the probability of an incorrect classification of a random element from the subset that has been assigned a random class within the subset. Information gain tells how much more accuracy a new decision node would add to the tree if chosen. GINI was chosen because of its popularity and its resource efficiency.

Decision Tree as a classifier was chosen because it is easy to implement and human-readable; also, decision trees can handle noisy data well because subsets without significance can be ignored by the algorithm that builds the tree. The classifier can be susceptible to overfitting, where the model becomes too specific to the data used to train it and provides poor results when used with new input data. Overfitting can become a problem when trying to apply the model to a mode-generalized dataset.

Random Forest

Random Forest [13] is an ensemble classifier, which tries to reduce the risk of overfitting a decision tree by constructing a collection of decision trees from random subsets in the data. The resulting collection of decision trees is smaller in depth, has a reduced degree of correlation between the subset's attributes, and thus has a lower risk of overfitting.

When given input data to label, the model utilizes all the generated trees, feeds the input data into all of them, and uses the average of the individual labels of the trees as the final label given to the input.

Extremely Randomized Trees

Extremely Randomized Trees [30] builds upon the Random Forest introduced above by taking the same principle of splitting the data into random subsets and building a collection of decision trees from these. In order to further randomize the decision trees, the attributes by which the splitting of the subsets is done are also randomized, resulting in a more computationally efficient model than Random Forest while still alleviating the negative effects of overfitting.

Bagging

Bagging [11] is an ensemble classification technique that tries to reduce the effects of overfitting a model by creating multiple smaller training sets from the initial set; in our study, it creates multiple decision trees from these sets. The sets are created by sampling the initial set uniformly and with replacements, which means that individual data points can appear in multiple training sets. The resulting trees can be used in labeling new input through a voting process by the trees.

AdaBoost

AdaBoost [28] is a classifier based on the concept of boosting. The implementation of the algorithm in this study uses a collection of decision trees, but new trees are created with the intent of correctly labeling instances of data that were misclassified by previous trees. For each round of training, a weight is assigned to each sample in the data. After the round, all misclassified samples are given higher priority in the subsequent rounds. When the number of trees reaches a predetermined limit or the accuracy cannot be improved further, the model is finished. When predicting the label of a new sample with the finished model, the final label is calculated from the weighted decisions of all the constructed trees. As Adaboost is based on decision trees, it can be resistant to overfitting and be more useful with generalized data. However, Adaboost is susceptible to noise data and outliers.

Gradient Boost

Gradient Boost [29] is similar to the other boosting methods. The algorithms for gradient boosting were developed by Jerome H. Friedman by expanding upon the ideas on arcing by Leo Breiman[12]. It uses a collection of weaker classifiers, which are created sequentially according to an algorithm. In the case of Gradient Boost as used in this study, the determining factor in building the new decision trees is the use of a loss function. The algorithm tries to minimize the loss function and, similarly to Adaboost, stops when the model has been fully optimized or the number of trees reaches the predetermined limit.

XGboost

XGBoost [17] is a scalable implementation of Gradient Boost. It was introduced by Tianqi Chen and Carlos Guestrin of University of Washington as a more resource efficient approach to boosting algorithm for large scale machine learning systems. The use of XGBoost could be seen to achieve performance resource improvements when constructing a model especially when building a model with large datasets. Since its introduction and

distribution as an open source package, XGBoost has been observed to affect the machine learning industry greatly. In the paper, Cien and Guestrin detail how XGBoost has been used in numerous winning solutions developed for machine learning competitions and challenges. Although the scale of this study isn't large enough to take advantage of XGBoost's scalability features, using it may provide good results.

3 RELATED WORK

3.1 Related Work

In this Section, we report on the most relevant works on pull requests.

3.1.1 Pull Request Process

Pull requests have been studied from different points of view, such as pull-based development [32], [31] and [68], usage of real online resources [74], pull requests reviewer assignment [73], and acceptance process [32], [56], [65], [39]. Another issue regarding pull requests that have been investigated is latency. Yu et al. [72] define latency as a complex issue related to many independent variables such as the number of comments and the size of a pull request.

Zampetti et al. [74] investigated how, why, and when developers refer to online resources in their pull requests. They focused on the context and real usage of online resources and how these resources have evolved during time. Moreover, they investigated the browsing purpose of online resources in pull request systems. Instead of investigating commit messages, they evaluated only the pull request descriptions, since generally the documentation of a change aims at reviewing and possibly accepting the pull request [32].

Yu et al. [73] worked on pull requests reviewer assignment in order to provide an automatic organization in GitHub that leads to an effort waste. They proposed a reviewer recommender, who should predict highly relevant reviewers of incoming pull requests based on the textual semantics of each pull request and the social relations of the developers. They found several factors that influence pull requests latency such as size, project age, and team size. The importance of social relationship between the pull request submitter and its reviewer is also supported by [66]

This approach reached a precision rate of 74% for top-1 recommendations, and a recall rate of 71% for top-10 recommendations. However, the authors did not consider the aspect of code quality. The results are confirmed also by [65].

Recent studies investigated the factors that influence the acceptance and rejection of a pull request.

There is no difference in treatment of pull-requests coming from the core team and from the community. Generally merging decision is postponed based on technical factors [33],[57]. Generally, pull requests that passed the build phase are generally merged more frequently [75]

Integrators decide to accept a contribution after analysing source code quality, code style, documentation, granularity, and adherence to project conventions [32]. Pull request's programming language had a significant influence on acceptance [56]. Higher acceptance was mostly found for Scala, C, C#, and R programming languages. Factors regarding developers are related to acceptance process, such as the number and experience level of developers [55], and the developers reputation who submitted the pull request [16]. Moreover, social connection between the pull-request submitter and project manager concerns the acceptance when the core team member is evaluating the pull-request [67].

Zhang et al. [77] studied how project maintainers and contributors handle multiple pull requests that compete with each other by proposing to change the same part of the source code simultaneously. They found that the majority of the studied Java projects, 45 out of 60, contained overlapping pull requests but concluded that the competition doesn't seem to affect the integration of the pull requests.

Rejection of pull requests can increase when technical problems are not properly solving and if the number of forks increase too [55]. Other most important rejection factors are inexperience with pull requests; the complexity of contributions; the locality of the artifacts modified; and the project's policy contribution [65].

From the integrator's perspective, social challenges that needed to be addressed, for example, how to motivate contributors to keep working on the project and how to explain the reasons of rejection without discouraging them. From the contributor's perspective, they found that it is important to reduce response time, maintain awareness, and improve communication [32].

Legay et al. [41] studied the impact of past contributions to the subsequent submitted pull requests. The preliminary results show that continued contribution to a project is correlated with higher pull request acceptance rates. The study also concluded that a developer whose pull request is rejected is less likely to try to contribute to the project again. Project maintainers should especially avoid alienating new contributors from contributing to the project.

3.1.2 Software Quality of Pull Requests

To the best of our knowledge, only few studies have focused on the quality aspect of pull request acceptance [32], [31], [39].

Gousios et al. [32] investigated the pull-based development process focusing on the factors that affect the efficiency of the process and contribute to the acceptance of a pull

request, and the related acceptance time. They analyzed the GHTorrent corpus and another 291 projects. The results showed that the number of pull requests increases over time. However, the proportion of repositories using them is relatively stable. They also identified common driving factors that affect the lifetime of pull requests and the merging process. Based on their study, code reviews did not seem to increase the probability of acceptance, since 84% of the reviewed pull requests were merged.

Gousios et al. [31] also conducted a survey aimed at characterizing the key factors considered in the decision-making process of pull request acceptance. Quality was revealed as one of the top priorities for developers. The most important acceptance factors they identified are: targeted area importance, test cases, and code quality. However, the respondents specified quality differently from their respective perception, as *conformance*, *good available documentation*, and *contributor reputation*.

Kononenko et al. [39] investigated the pull request acceptance process in a commercial project addressing the quality of pull request reviews from the point of view of developers' perception. They applied data mining techniques on the project's GitHub repository in order to understand the merge nature and then conducted a manual inspection of the pull requests. They also investigated the factors that influence the merge time and outcome of pull requests such as pull request size and the number of people involved in the discussion of each pull request. Developers' experience and affiliation were two significant factors in both models. Moreover, they report that developers generally associate the quality of a pull request with the quality of its description, its complexity, and its revertability. However, they did not evaluate the reason for a pull request being rejected. These studies investigated the software quality of pull requests focusing on the trustworthiness of developers' experience and affiliation [39]. Moreover, these studies did not measure the quality of pull requests against a set of rules, but based on their acceptance rate and developers' perception. Our work complements these works by analyzing the code quality of pull requests in popular open-source projects and how the quality, specifically issues in the source code, affect the chance of a pull request being accepted when it is reviewed by a project maintainer. We measured code quality against a set of rules provided by PMD, one of the most frequently used open-source software tools for analyzing source code.

4 CASE STUDY DESIGN

4.1 Case Study Design

We designed our empirical study as a case study based on the guidelines defined by Runeson and Höst [59]. In this Section, we describe the case study design, including the goal and the research questions, the study context, the data collection, and the data analysis procedure.

4.1.1 Goal and Research Questions

The goal of this work is to investigate the role of code quality in pull request acceptance.

Accordingly, to meet our expectations, we formulated the goal as follows, using the Goal/Question/Metric (GQM) template [7]:

Purpose Analyze
Object the acceptance of pull requests
Quality with respect to their code quality
Viewpoint from the point of view of developers
Context in the context of Java projects

Based on the defined goal, we derived the following Research Questions (**RQs**):

RQ1 What is the distribution of TD items violated by the pull requests in the analyzed software systems?

RQ2 Does code quality affect pull request acceptance?

RQ3 Does code quality affect pull request acceptance considering different types and levels of severity of TD items?

RQ1 aims at assessing the distribution TD items violated by pull requests in the analyzed software systems. We also took into account the distribution of TD items with respect to

their priority level as assigned by PMD (P1-P5). These results will also help us to better understand the context of our study.

RQ2 aims at finding out whether the project maintainers in open-source Java projects consider quality issues in the pull request source code when they are reviewing it. If code quality issues affect the acceptance of pull requests, the question is what kind of TD items errors generally lead to the rejection of a pull request.

RQ3 aims at finding out if a severe code quality issue is more likely to result in the project maintainer rejecting the pull request. This will allow us to see whether project maintainers should pay more attention to specific issues in the code and make code reviews more efficient.

4.1.2 Context

The projects for this study were selected using "criterion sampling" [52]. The criteria for selecting projects were as follows:

- Uses Java as its primary programming language
- Older than two years
- Had active development in last year
- Code is hosted on GitHub
- Uses pull requests as a means of contributing to the code base
- Has more than 100 closed pull requests

Moreover, we tried to maximize diversity and representativeness considering a comparable number of projects with respect to project age, size, and domain, as recommended by Nagappan et al. [48].

We selected 28 projects according to these criteria. The majority, 22 projects, were selected from the Apache Software Foundation repository[1]. The repository proved to be an excellent source of projects that meet the criteria described above. This repository includes some of the most widely used software solutions, considered industrial and mature, due to the strict review and inclusion process required by the ASF. Moreover, the included projects have to keep on reviewing their code and follow a strict quality process[2].

The remaining six projects were selected with the help of the Trending Java repositories list that GitHub provides[3]. GitHub provides a valuable source of data for the study of code reviews [36].

[1] http://apache.org
[2] https://incubator.apache.org/policy/process.html
[3] https://github.com/trending/java

Table 4.1. *Selected projects*

Project Owner/Name	**#PR**	**Time Frame**	**#LOC**
apache/any23	129	2013/12-2018/11	78.351
apache/dubbo	1,270	2012/02-2019/01	133.633
apache/calcite	873	2014/07-2018/12	337.436
apache/cassandra	182	2018/10-2011/09	411.248
apache/cxf	455	2014/03-2018/12	807.517
apache/flume	180	2012/10-2018/12	103.706
apache/groovy	833	2015/10-2019/01	396.433
apache/guacamole-client	331	2016/03-2018/12	65.928
apache/helix	284	2014/08-2018/11	191.832
apache/incubator-heron	2,191	2015/12-2019/01	207.364
hibernate/hibernate-orm	2,573	2010/10-2019/01	797.303
apache/kafka	5,522	2013/01-2018/12	376.683
apache/lucene-solr	264	2016/01-2018/12	1.416.200
apache/maven	166	2013/03-2018/12	107.802
apache/metamodel	198	2014/09-2018/12	64.805
mockito/mockito	726	2012/11-2019/01	57.405
apache/netbeans	1,026	2017/09-2019/01	6.115.974
netty/netty	4,129	2010/12-2019/01	275.970
apache/opennlp	330	2016/04-2018/12	136.545
apache/phoenix	203	2014/07-2018/12	366.588
apache/samza	1,475	2014/10-2018/10	129.282
spring-projects/spring-framework	1,850	2011/09-2019/01	717.962
spring-projects/spring-boot	3,076	2013/06-2019/01	348.091
apache/storm	2,863	2013/12-2018/12	359.906
apache/tajo	1,020	2014/03-2018/07	264.790
apache/vxquery	169	2015/04-2017/08	264.790
apache/zeppelin	3,194	2015/03-2018/12	218.956
openzipkin/zipkin	1,474	2012/06-2019/01	121.507
Total	**36,344**		**14.683.977**

In the selection, we manually selected popular Java projects using the criteria mentioned before.

In Table 4.1, we report the list of the 28 projects that were analyzed along with the number of pull requests ("*#PR*"), the time frame of the analysis, and the size of each project (*"#LOC")*.

Table 4.2. Example of data structure used for the analysis

		Dependent Variable	Independent Variables		
Project ID	PR ID	Accepted PR	Rule1	...	Rule n
Cassandra	ahkji	1	0		3
Cassandra	avfjo	0	0		2

4.1.3 Data Collection

We first extracted all pull requests from each of the selected projects using the GitHub REST API v3 [4].

For each pull request, we fetched the code from the pull request's branch and analyzed the code using PMD. The default Java rule set for PMD was used for the static analysis. We filtered the TD items added in the main branch to only include items introduced in the pull request. The filtering was done with the aid of a diff-file provided by GitHub API and compared the pull request branch against the master branch.

We identified whether a pull request was accepted or not by checking whether the pull request had been marked as merged into the master branch or whether the pull request had been closed by an event that committed the changes to the master branch. Other ways of handling pull requests within a project were not considered.

4.1.4 Data Analysis

The result of the data collection process was a csv file reporting the dependent variable (pull request accepted or not) and the independent variables (number of TD items introduced in each pull request). Table 4.2 provides an example of the data structure we adopted in the remainder of this work.

For **RQ1**, we first calculated the total number of pull requests and the number of TD items present in each project. Moreover, we calculated the number of accepted and rejected pull requests. For each TD item, we calculated the number of occurrences, the number of pull requests, and the number of projects where it was found. Moreover, we calculated descriptive statistics (average, maximum, minimum, and standard deviation) for each TD item.

In order to understand if TD items affect pull request acceptance (**RQ2**), we first determined whether there is a significant difference between the expected frequencies and the observed frequencies in one or more categories. First, we computed the χ^2 test.

Then, we selected eight Machine Learning techniques and compared their accuracy. To overcome to the limitation of the different techniques, we selected and compared eight

[4] https://developer.github.com/v3/

Table 4.3. *Accuracy measures*

Accuracy Measure	Formula
Precision	$\frac{TP}{FP+TP}$
Recall	$\frac{TP}{FN+TP}$
MCC	$\frac{TP*TN-FP*FN}{\sqrt{(FP+TP)(FN-TP)(FP+TN)(FN+TN)}}$
F-measure	$2*\frac{precision*recall}{precision+recall}$

TP: True Positive; TN: True Negative; FP: False Positive; FN: False Negative

of them. The description of the different techniques, and the rationale adopted to select each of them is reported in Section 2.

χ^2 test could be enough to answer our RQs. However, in order to support possible follow-up of the work, considering other factors such as LOC as independent variable, Machine Learning techniques can provide much more accuracy results.

We examined whether considering the priority value of an issue affects the accuracy metrics of the prediction models (**RQ3**). We used the same techniques as before but grouped all the TD items in each project into groups according to their priorities. The analysis was run separately for each project and each priority level (28 projects * 5 priority level groups) and the results were compared to the ones we obtained for RQ2. To further analyze the effect of issue priority, we combined the TD items of each priority level into one data set and created models based on all available items with one priority.

Once a model was trained, we confirmed that the predictions about pull request acceptance made by the model were accurate (**Accuracy Comparison**). To determine the accuracy of a model, 5-fold cross-validation was used. The data set was randomly split into five parts. A model was trained five times, each time using four parts for training and the remaining part for testing the model. We calculated accuracy measures (Precision, Recall, Matthews Correlation Coefficient, and F-Measure) for each model (see Table 4.3) and then combined the accuracy metrics from each fold to produce an estimate of how well the model would perform.

We started by calculating the commonly used metrics, including F-measure, precision, recall, and the harmonic average of the latter two. Precision and recall are metrics that focus on the true positives produced by the model. Powers [54] argues that these metrics can be biased and suggests that a contingency matrix should be used to calculate additional metrics to help understand how negative predictions affect the accuracy of the constructed model. Using the contingency matrix, we calculated the model's Matthew Correlation Coefficient (MCC), which suggests as the best way to reduce the information provided by the matrix into a single probability describing the model's accuracy [54].

For each classifier to easily gauge the overall accuracy of the machine learning algorithm in a model [9], we calculated the Area Under The Receiver Operating Characteristic (AUC). For the AUC measurement, we calculated Receiver Operating Characteristics

(ROC) and used these to find out the AUC ratio of the classifier, which is the probability of the classifier ranking a randomly chosen positive higher than a randomly chosen negative one.

4.1.5 Replicability

In order to allow our study to be replicated, we have published the complete raw data in the replication package[5].

[5]https://figshare.com/s/d47b6f238b5c92430dd7

5 IMPLEMENTATION

In this Section the implementation of the program used to collect the data for this study is discussed. First the reader is given some background information about the solutions used and then a detailed overview of the whole process is presented.

5.1 Software structure and design decisions

The data collection and analysis script used in this study is built as a Node.js application with Typescript running inside a docker container that uses external Java and Python programs to extract data from GitHub Pull Requests and analyze that data with machine learning techniques to create results for the study described in this thesis.

5.1.1 Docker

Docker is a software used to manage and run container images inside the host system. A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another[1]. The use of containers is made possible by Linux Containers project (and later Windows Containers) which aims to allow the user to run an isolated process environment that can access computer's resources without needing to run a fully fledged virtual machine. This process allows more resource-efficient virtualization and is quickly becoming an industry standard in running software with its dependencies in a standard environment independent of the host machine's configuration or operating system.

Docker was chosen as the environment to run the software in because of the multiple operating system specific dependencies required to run all the parts of the analysis sequence. With a standardized environment in which to run the analysis, the software could be used in multiple different machines simultaneously. This ensured that the analysis could be done in a more reasonable time frame and that the software could be distributed to and used by other researchers without extensive setup of dependencies.

[1] https://www.docker.com/resources/what-container

5.1.2 Node.js

Node.js is a Javascript runtime solution that uses a popular V8 Javascript engine to run user code in various different operating systems[2]. Traditionally interpreting Javascript code was thought to be the web browser's task, but Javascript runtimes for desktop and server applications have become popular with due to their ease of use and the ability to use a single programming language for both websites and the server applications serving them.

Node.js was chosen as the basis of the software primarily because the author is familiar with Javascript language and the node ecosystem. Potential performance issues commonly associated with Javascript and other interpreted languages were not taken into account because the bulk of the running time of the application is taken by the GitHub API network requests and the external analysis software

5.1.3 Typescript

Typescript[3] is a superset of Javascript designed by Microsoft to aid in software development by giving the user access to strong type definitions while still remaining compatible with Javascript based runtimes and engines due to the Typescript compiler compiling source code written in Typescript back to Javascript. It was first introduced in 2012 and matured into a stable release by April 2014 [4]. Since then, Typescript's popularity has risen sharply among javascript developers in recent years[5] with the most important reasons being "Robust, less error-prone code", "Elegant programming style & patterns" and "Powerful developer tooling".

Typescript was chosen because it helps to improve code quality through a strong typing system, provides new features compared to using plain Javascript and has good code editor support to provide faster development time thanks to the additional information that the typing system can provide to the tooling.

5.1.4 Machine Learning with Python

The machine learning was implemented using Python programming language and with the help of established machine learning libraries such as scikit-learn and XGBoost. Python provides the user with easy tools to handle the large datasets that machine learning tasks can require through the use of NumPy-library. In order to speed up the operation of building the machine learning models, multiprocessing capabilities of Python were

[2] https://nodejs.org/en/about/
[3] https://www.typescriptlang.org/
[4] https://devblogs.microsoft.com/typescript/announcing-typescript-1-0/
[5] https://2018.stateofjs.com/javascript-flavors/typescript/

```
FROM node:11.2.0

# install dependencies
RUN apt-get update
RUN apt-get -y install bash gcc git gfortran python3 \
  python3-pip build-essential unzip zip wget libpng-dev \
  openjdk-8-jre libopenblas-dev sqlite python3-dev

RUN pip3 install --upgrade setuptools
RUN pip3 install numpy scipy pandas matplotlib \
  scikit-learn xgboost joblib

RUN git config --system core.longpaths true
ENV LC_ALL=C.UTF-8

WORKDIR /app

# fetch PMD, unzip it, link the runscript
RUN wget https://github.com/pmd/pmd/releases/download/ \
  pmd_releases%2F6.9.0/pmd-bin-6.9.0.zip

RUN unzip pmd-bin-6.9.0.zip
RUN ln -s /app/pmd-bin-6.9.0/bin/run.sh /usr/local/bin/

# install the npm packages and copy the sources
COPY /package*.json ./
RUN npm install --no-package-lock
COPY . ./

CMD ["/bin/bash"]
```

Listing 5.1. *Docker container definition*

used. This allowed the analysis script to take full advantage of the machine's processing cores and do data operations concurrently.

5.2 Software structure and considerations

The structure of the software implementation is shown in figure 5.1. The figure also depicts the flow of control within the process starting with the "Entry point" to the docker container. The docker container image definition is shown in listing 5.1. The container created from that image will have all the necessary dependencies installed and can be reused for each of the project in the study. When the actual data processing is started, the script starts to download the given project's pull request data through the GitHub API and starts to parse the data to determine whether a pull request was accepted or rejected.

The pull requests and the customizable nature of their use in GitHub posed an interesting

challenge where there is no one simple way of determining the acceptance of a pull request. However, two main ways of accepting a pull request and adding the code in to the master branch of a project were detected:

1. Accepting the pull request through the GitHub web interface
2. Accepting the pull request with a pull request closing commit event

The first method is straightforward. The GitHub API will provide the user with a timestamp of the merging and the code is merged as it appears in the original pull request. Unfortunately this method of accepting pull requests is seldom used as maintainers might prefer to have more control over the merging process such as using rebasing instead of merging.

In the projects analyzed in this study a far more common method was the second method, which consist of the maintainer manually applying the new source code in to the master branch of the repository and then submitting the change in a closing event to the pull request. This type of an event can then be parsed from the pull request data. However, a closing event with a commit doesn't guarantee that the code added is just as the pull request creator submitted it. In order to verify that in the event of an accepted closing commit the code is unchanged by the maintainer, the script also checks if the exact commits in the pull request are present in the master branch or if the author of the closing commit is indeed the author of the pull request.

Once the pull request details are gathered and saved into the database, the script uses Git to fetch the external pull request branch using the pull request id and changes the current active branch to that one. This allows PMD the access to the source code files. Before the static code analysis can take place, a list of changed files in the pull request branch is compiled with the help of a diff file that Git provides. A diff file lists the the additions and deletions of code between two branches, in this case the master branch and the pull request branch. The changed files are sent to PMD for analysis and the resulting lists of issues are filtered to only include the issues that are introduced by the code in the pull request. This step of the process is shown in abbreviated form in Listing 5.3

The stored issues in the database are then linked to the parsed pull requests by counting the occurrences of a particular type of issue. This gives us a table that can be exported into a comma separated file that the machine learning script can use. Before the creation of machine learning models, the list of pull requests is further filtered to exclude pull requests that don't have any PMD detected issues in them. The filtering is done to reduce the amount of noise in the dataset and help produce more accurate results.

The last step in the process is to construct the machine learning models with the test data and then test their accuracies with the test data. The machine learning script takes the whole dataset and creates a predetermined amount of training and testing sets from it using scikit-learn package's StratifiedKFold. A model is trained and tested for each fold and the key metrics are recorded into memory. When all of the folds have been used,

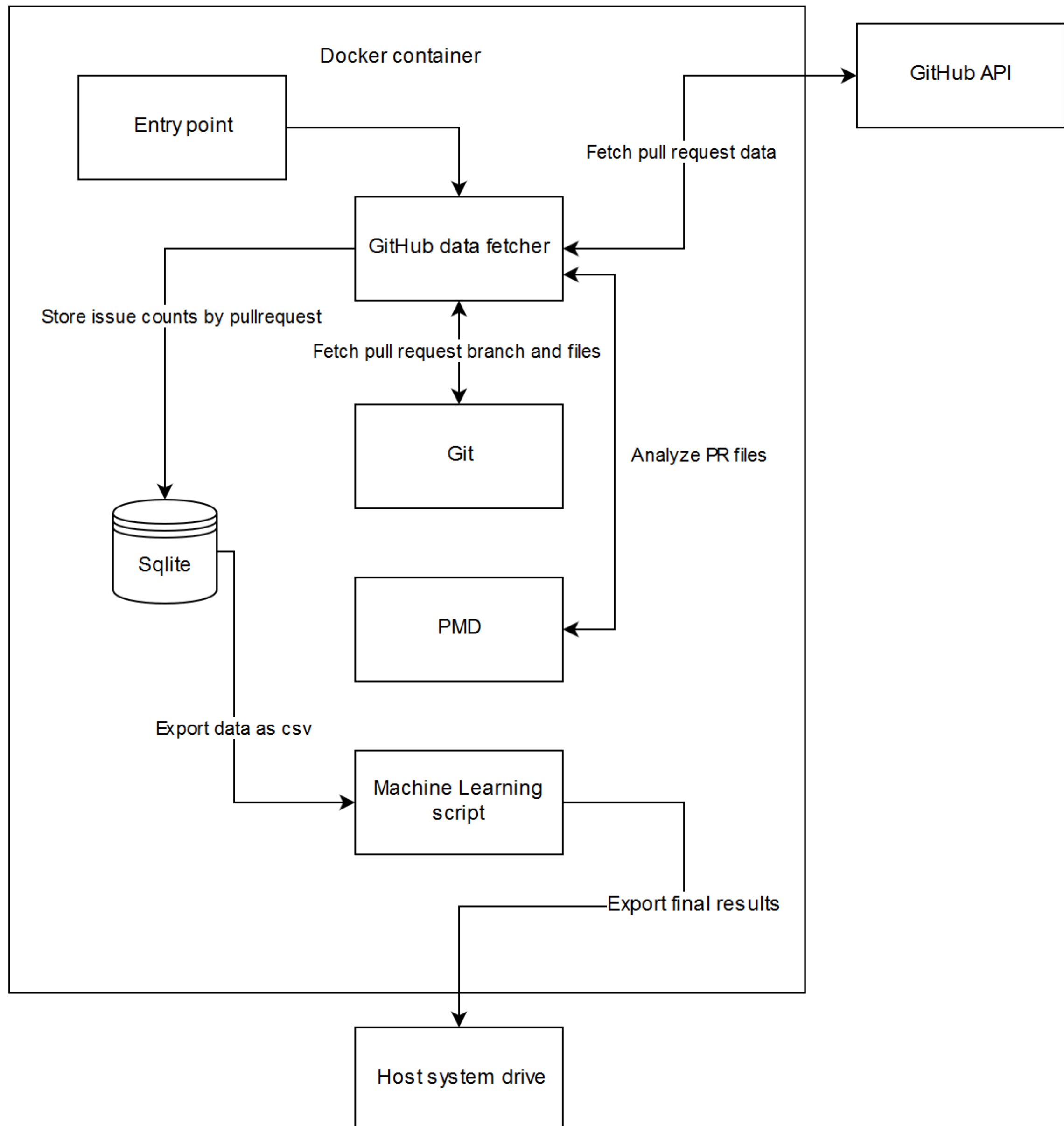


Figure 5.1. *The structure of the software implemented*

an avarage of all individual metric types are calculated and saved into a CSV file. This process of training, testing and logging of metrics is repeated for each of the eight classifiers used in this study. Once all of the data is collected and calculated, the final results are exported out of the docker container into a mounted folder on the host computer's filesystem. The main function in Listing 5.2 shows the scripts overall execution process and exporting of the results into a mounted result-folder.

```
1  import child_process from 'child_process';
2  // execSync spawns a child process within the script
3  // that is run synchronously so the script
4  // waits for these operations to complete
5  // before continuing
6  const execSync = child_process.execSync;
7
8  async function main() {
9    await fetchAndProcessPR();
10   await analyzePRs();
11   await syncIssuesToPRTable();
12   await countIssues();
13   // export the pull request data
14   execSync(
15     'sqlite3 -header -csv database.sqlite \
16     "select * from pullrequest where analyzed = 1;" \
17     > pullrequest.csv'
18   );
19   // run python analysis and export the results
20   logger.info("Running analysis on issue counts");
21   execSync("python3 analysis_script.py", { stdio: [] });
22   execSync(`zip ${projectName}_result \
23     ./added_removed_total_comparisons/*`);
24   execSync(`mv ${projectName}_result.zip ./result`);
25   execSync(`zip ${projectName}_data \
26     pullrequest.csv issue.csv commit.csv`);
27   execSync(`mv ${projectName}_data.zip ./result`);
28   logger.info(`Process finished, result copied to the volume`);
29 }
```

Listing 5.2. *Example of the main function*

```
1  async function analyzePRs() {
2    let errorCollection = [];
3    let pmdResult = null;
4    for (const one of pullrequests) {
5      // parsed
6      const diffjson = JSON.parse(one.diffanalysis);
7      // go over the diff one file at a time
8      for (const filediff of diffjson) {
9        if (!filediff || !filediff.to || !filediff.chunks) {
10         continue;
11       }
12       // check that the file contains java code
13       const name = last(filediff.to.split("/"));
14       if (!name || last(name.split(".")) !== "java") {
15         continue;
16       }
17       // find the full path in the filesystem
18       const filepath = execSync(
19         `find ${__dirname}/projects/${projectName} \
20         -path *${filediff.to}`,
21         {
22           stdio: [2]
23         }
24       ).toString("utf8").trim();
25       if (!filepath) { continue; }
26       // run PMD on the file
27       pmdResult = execSync(
28         `run.sh pmd -d ${filepath} -f csv -R ruleset.xml \
29         -failOnViolation false -no-cache -l java`,{ stdio: [2] }
30       );
31       const fileissues = parse(pmdResult, {
32         columns: true,
33         relax_column_count: true
34       });
35       let addedLines = [];
36       // add all new lines introduces in the file
37       // chunks into an array
38       filediff.chunks.forEach(c => {
39         const chunkAddedLines = c.changes
40           .filter(x => x.type === "add")
41           .map(y => y.ln.toString());
42         addedLines = addedLines.concat(chunkAddedLines);
43       });
44       // take only issues introduced in the pull request code
45       const matchedErrors =
46         fileissues.filter(x => addedLines.includes(x.Line));
47       errorCollection = errorCollection.concat(matchedErrors);
48     }
49   }
50 }
```

Listing 5.3. *Example of PMD analysis*

6 RESULTS

RQ1. What is the distribution of TD items violated by the pull requests in the analyzed software systems?

For this study, we analyzed 36,344 pull requests violating 253 TD items and contained more than 4.7 million times (Table 6.1) in the 28 analyzed projects. We found that 19,293 pull requests (53.08%) were accepted and 17,051 pull requests (46.92%) were rejected. Eleven projects contained the vast majority of the pull requests (80%) and TD items (74%). The distribution of the TD items differs greatly among the pull requests. For example, the projects *Cassandra* and *Phoenix* contain a relatively large number of TD items compared to the number of pull requests, while *Groovy*, *Guacamole*, and *Maven* have a relatively small number of TD items.

Taking into account the priority level of each rule, the vast majority of TD items (77.86%) are classified with priority level 3, while the remaining ones (22.14%) are equally distributed among levels 1, 2, and 4. None of the projects we analyzed had any issues rated as priority level 5.

Table 6.2 reports the number of TD items (”*#TD item*”) and their number of occurrences (”*#occurrences*”) grouped by priority level (”*Priority*”).

Table 6.2. *Distribution of TD items in pull requests - (RQ1)*

Priority	#TD Items	#occurrences	% PR Acc.	% PR Rej.
All	253	4,703,146	96.05	100.00
4	18	85,688	77.78	100.00
3	197	4,488,326	96.95	100.00
2	22	37,492	95.45	95.45
1	16	91,640	100.00	100.00

Looking at the TD items that could play a role in pull request acceptance or rejection, 243 of the 253 TD items (96%) are present in both cases, while the remaining 10 are found only in cases of rejection (Table 6.2).

Table 6.1. *Distribution of pull requests (PR) and technical debt items (TD items) in the selected projects - (RQ1)*

Project Name	#PR	#TD Items	% Acc.	% Rej.
apache/any23	129	11,573	90.70	9.30
apache/dubbo	1,270	169,751	52.28	47.72
apache/calcite	873	104,533	79.50	20.50
apache/cassandra	182	153,621	19.78	80.22
apache/cxf	455	62,564	75.82	24.18
apache/flume	180	67,880	60.00	40.00
apache/groovy	833	25,801	81.39	18.61
apache/guacamole-client	331	6,226	92.15	7.85
apache/helix	284	58,586	90.85	9.15
apache/incubator-heron	2,191	138,706	90.32	9.68
hibrenate/hibernate-orm	2,573	490,905	16.27	83.73
apache/kafka	5,522	507,423	73.51	26.49
apache/lucene-solr	264	72,782	28.41	71.59
apache/maven	166	4,445	32.53	67.47
apache/metamodel	198	25,549	78.28	21.72
mockito/mockito	726	57,345	77.41	22.59
apache/netbeans	1,026	52,817	83.14	16.86
netty/netty	4,129	597,183	15.84	84.16
apache/opennlp	330	21,921	82.73	17.27
apache/phoenix	203	214,997	9.85	90.15
apache/samza	1,475	96,915	69.52	30.48
spring-projects/spring-framework	1,850	487,197	15.68	84.32
spring-projects/spring-boot	3,076	156,455	8.03	91.97
apache/storm	2,863	379,583	77.96	22.04
apache/tajo	1,020	232,374	67.94	32.06
apache/vxquery	169	19,033	30.77	69.23
apache/zeppelin	3,194	408,444	56.92	43.08
openzipkin/zipkin	1,474	78,537	73.00	27.00

Focusing on TD items that have with a ”double role”, we analyzed the distribution in each case. We discovered that 88 TD items have a diffusion rate of more than 60% in the case of acceptance and 127 have a diffusion rate of more than 60% in the case of rejection.

The remaining 38 are equally distributed.

Table 6.3 presents preliminary information related to the twenty most recurrent TD items. We report descriptive statistics by means of Average ("*Avg.*"), Maximum ("*Max*"), Minimum ("*Min*"), and Standard Deviation ("*Std. dev.*"). Moreover, we include the priority of each TD item ("*Priority*"), the sum of issue rows of that rule type found in the issues master table ("*# Total occurrences*"), and the number of projects in which the specific TD item has been violated (*"#Project"*).

The complete list is available in the replication package (Section 4.1.5).

Table 6.3. *Descriptive statistics (the 20 most recurrent TD items) - (RQ1)*

TD Item	Prio.	#occur.	#PR	#prj.	Avg	Max	Min	Std. dev.
LawOfDemeter	4	1,089,110	15,809	28	38,896.78	140,870	767	40,680.62
MethodArgumentCouldBeFinal	4	627,688	12,822	28	22,417.42	105,544	224	25,936.63
CommentRequired	4	584,889	15,345	28	20,888.89	66,79	39	21,979.94
LocalVariableCouldBeFinal	4	578,760	14,920	28	20,670	67394	547	20,461.61
CommentSize	4	253,447	11,026	28	9,051.67	57,074	313	13,818.66
JUnitAssertionsShould-IncludeMessage	4	196,619	6,738	26	7,562.26	38,557	58	10822.38
BeanMembersShouldSerialize	4	139,793	8,865	28	4,992.60	22,738	71	5,597.45
LongVariable	4	122,881	8,805	28	4,388.60	19,958	204	5,096.23
ShortVariable	4	112,333	7,421	28	4,011.89	21,900	26	5,240.06
OnlyOneReturn	4	92,166	7,111	28	3,291.64	14,163	42	3,950.45
CommentDefaultAccessModifier	4	58,684	5,252	28	2,095.85	12,535	6	2,605.75
DefaultPackage	4	42,396	4,201	28	1,514.14	9,212	2	1,890.76
ControlStatementBraces	4	39,910	2,689	27	1,478.14	11,130	1	2,534.29
JUnitTestContainsTooMany-Asserts	4	3,6022	4,954	26	1,385.46	7,888	7	1,986.52
AtLeastOneConstructor	4	29,516	5,561	28	1,054.14	6,514	21	1,423.12
UnnecessaryFullyQualifiedName	4	27,402	1,393	27	1,014.88	7469	5	1,742.19
AvoidDuplicateLiterals	4	27,224	3,748	28	972.28	3,595	15	1,053.63
SignatureDeclareThrows-Exception	4	26,188	3,049	27	969.92	5,734	5	1,512.59
AvoidInstantiating-ObjectsInLoops	3	25,344	3,626	28	905.14	5,338	30	1,219.46
FieldNamingConventions	3	25,062	2,724	28	895.07	4,505	6	1,035.85

Summary of RQ1

Among the 36,344 analyzed pull requests, we discovered 253 different type of TD items (PMD Rules) violated more that 4.7 million times. Nearly half of the pull requests had been accepted and the other half had been rejected. 243 of the 253 TD items were found to be present in both cases. The vast majority of these TD items (197) have priority level 3.

RQ2. Does code quality affect pull request acceptance?

To answer this question, we trained machine learning models for each project using all possible pull requests at the time and using all the different classifiers introduced in Section 2. A pull request was used if it contained Java that could be analyzed with PMD.

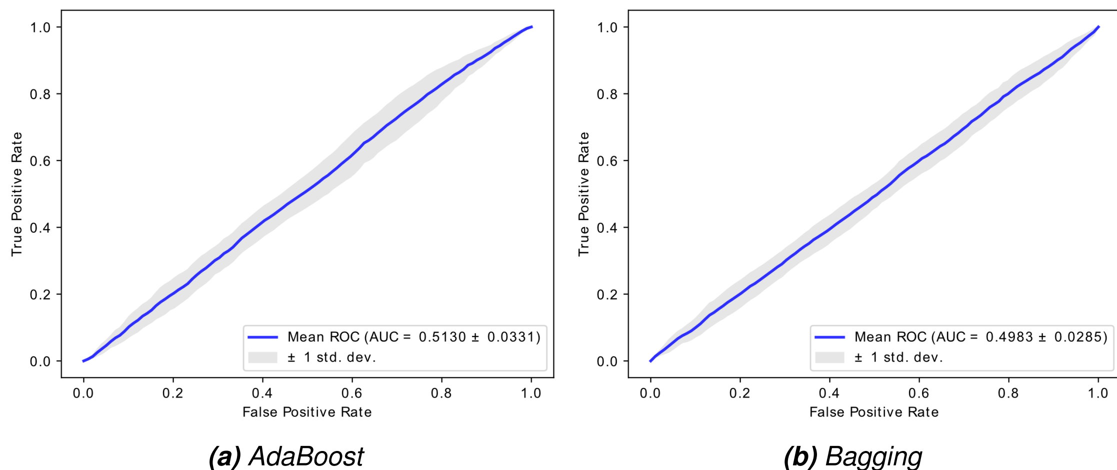
There are some projects in this study that are multilingual, so filtering of the analyzable pull requests was done out of necessity.

Once we had all the models trained, we tested them and calculated the accuracy measures described in Table 4.3 for each model. We then averaged each of the metrics from the classifiers for the different techniques. The results are presented in Table 6.4. The averaging provided us with an estimate of how accurately we could predict whether maintainers accepted the pull request based on the number of different TD items it has.

Table 6.4. *Model reliability - (RQ2)*

	Average between 5-fold validation models							
Accuracy Measure	**Logistic Regression**	**Decision Tree**	**Bagging**	**Random Forest**	**ExtraTrees**	**AdaBoost**	**Gradient Boosting**	**XGBoost**
AUC	50.91	50.12	49.83	50.75	50.54	51.30	50.64	50.92
Precision	49.53	48.40	48.56	49.33	49.20	48.74	49.30	49.20
RECALL	62.46	47.45	47.74	48.07	47.74	51.82	41.80	41.91
MCC	0.0235	-0.0020	0.0002	0.0135	0.0121	0.0045	0.0020	-0.0002
F-Measure	0.5514	0.4785	0.4795	0.4846	0.4826	0.4994	0.4416	0.4403

Figure 6.1. *ROC Curves of Adaboost and Bagging - (RQ2)*



(a) *AdaBoost* ***(b)*** *Bagging*

The results of this analysis are presented in Table 6.5. For reasons of space, we report only the most frequent 20 TD items. The table also contains the number of distinct PMD rules that the issues of the project contained. The rule count can be interpreted as the number of different types of issues found.

With almost all of the models' AUC for every method of prediction hovering around 50%, overall code quality does not appear to be a factor in determining whether a pull request is accepted or rejected.

There were some projects that showed some moderate success, but these can be dismissed as outliers.

We computed the χ^2 test on the contingency matrix (Table 6.6), obtaining a value of 0.12. This confirms the above results that the presence of TD items does not affect

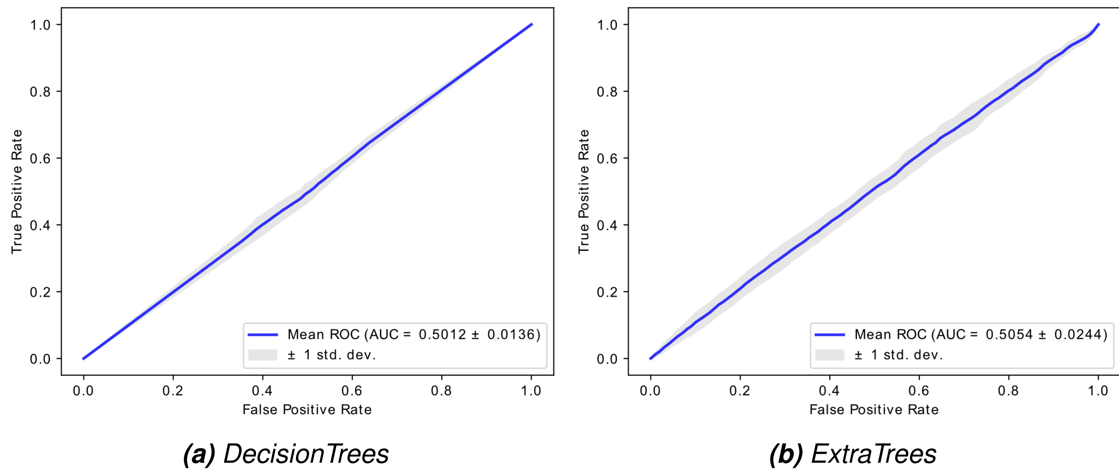
Table 6.5. *Summary of the quality rules related to pull request acceptance - (RQ2 and RQ3)*

Rule ID	Priority	#projects	#occur.	Importance (%)							
				Ada boost	Bagging	Decision Tree	Extra Trees	Gradient Boost	Logistic Regression	Random Forest	XG Boost
LawOfDemeter	4	28	1089110	0.12	-0.51	0.77	-0.74	-0.29	-0.09	-0.66	0.02
MethodArgument-CouldBeFinal	4	28	627688	-0.31	0.38	0.14	0.03	-0.71	-0.25	0.24	0.07
CommentRequired	4	28	584889	-0.25	-0.11	0.07	-0.30	-0.47	-0.17	0.58	-0.31
LocalVariable-CouldBeFinal	4	28	578760	-0.13	-0.20	0.55	0.28	0.08	-0.05	0.61	-0.05
CommentSize	4	28	253447	-0.24	-0.15	0.49	-0.08	-0.17	-0.05	-0.10	0.05
JUnitAssertions-ShouldInclude-Message	4	26	196619	-0.41	-0.84	0.22	-0.28	-0.19	-0.10	-0.75	0.14
BeanMembers-ShouldSerialize	4	28	139793	-0.33	-0.09	-0.03	-0.38	-0.37	0.17	0.26	0.07
LongVariable	4	28	122881	0.08	-0.19	-0.02	-0.25	-0.28	0.08	0.24	0.02
ShortVariable	4	28	112333	-0.51	-0.24	0.09	-0.04	-0.04	0.07	-0.25	-0.54
OnlyOneReturn	4	28	92166	-0.69	-0.03	0.02	-0.25	-0.08	-0.06	0.06	-0.13
CommentDefault-AccessModifier	4	28	58684	-0.17	-0.07	0.30	-0.41	-0.25	0.23	0.18	-0.10
DefaultPackage	4	28	42396	-0.37	-0.05	0.20	-0.23	-0.93	0.10	-0.01	-0.54
ControlStatement-Braces	4	27	39910	-0.89	0.09	0.58	0.29	-0.37	-0.03	0.08	0.25
JUnitTestContains-TooManyAsserts	4	26	36022	0.40	0.22	-0.25	-0.33	0.01	0.16	0.10	-0.17
AtLeastOne-Constructor	4	28	29516	0.00	-0.29	-0.06	-0.18	-0.19	-0.07	0.15	-0.22
UnnecessaryFully-QualifiedName	4	27	27402	0.00	0.08	0.25	-0.05	0.00	0.00	0.26	-0.11
AvoidDuplicate-Literals	4	28	27224	-0.20	0.05	0.33	-0.28	0.12	0.20	0.09	0.07
SignatureDeclare-ThrowsException	4	27	26188	-0.18	-0.10	0.04	-0.13	-0.05	0.11	0.33	-0.17
AvoidInstantiating-ObjectsInLoops	3	28	25344	-0.05	0.07	0.43	-0.14	-0.27	-0.13	0.52	-0.07
FieldNaming-Conventions	3	28	25062	0.09	0.00	0.16	-0.21	-0.10	-0.01	0.07	0.19

Table 6.6. *Contingency matrix*

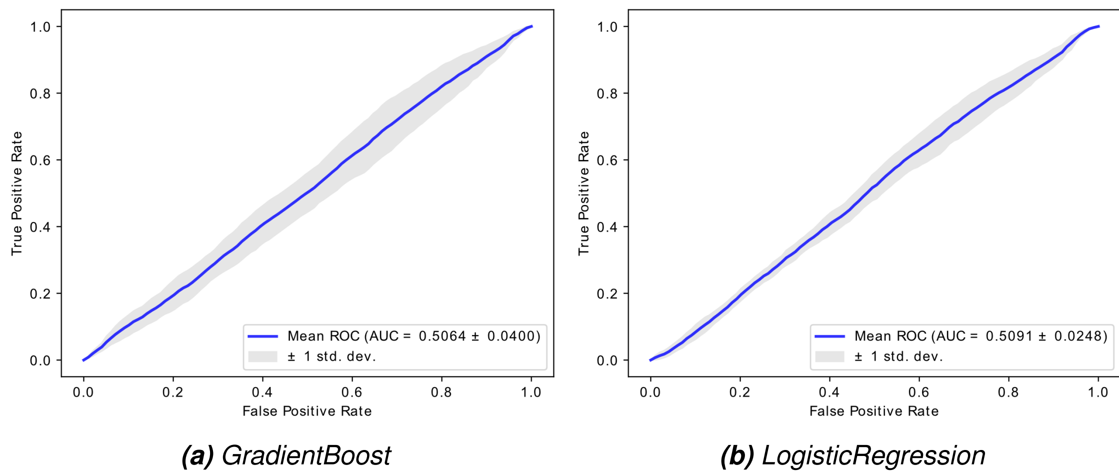
	TD items	No TD items
PR accepted	10.563	8.558
PR rejected	11.228	5.528

Figure 6.2. *ROC Curves of DecitionTrees and ExtraTrees - (RQ2)*



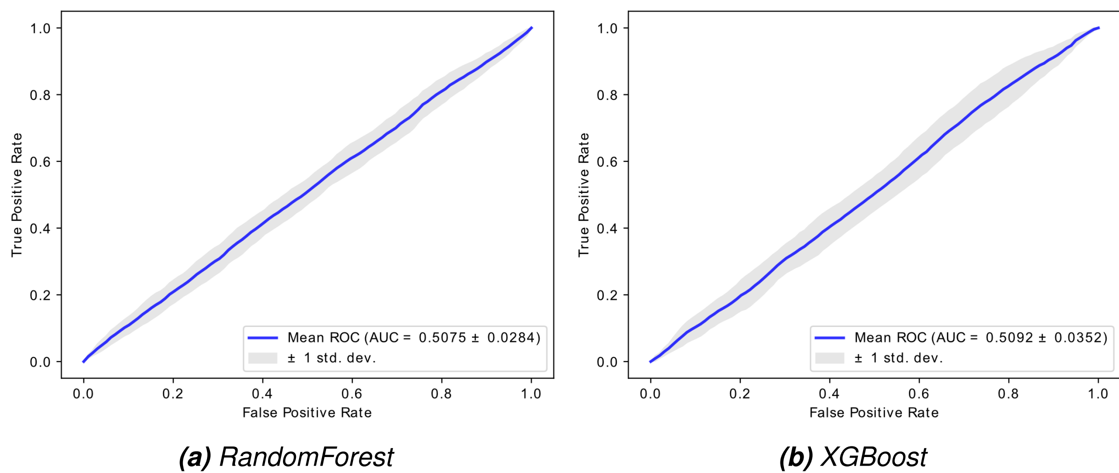
(a) *DecisionTrees* ***(b)*** *ExtraTrees*

Figure 6.3. *ROC Curves of GradientBoost and LogisticRegression - (RQ2)*



(a) *GradientBoost* ***(b)*** *LogisticRegression*

Figure 6.4. *ROC Curves of RandomForest and XGBoost - (RQ2)*



(a) *RandomForest* ***(b)*** *XGBoost*

pull request acceptance (which means that TD items and pull request acceptance are

mutually independent).

RQ3. Does code quality affect pull request acceptance considering different types and levels of severity of TD items?

To answer this research question, we introduced PMD priority values assigned to each TD item. By taking these priorities into consideration, we grouped all issues by their priority value and trained the models using data composed of only issues of a certain priority level.

Once we had run the training and tested the models with the data grouped by issue priority, we calculated the accuracy metrics mentioned above. These results enabled us to determine whether the prevalence of higher-priority issues affects the accuracy of the models. The affect on model accuracy or *importance* is determined with the use of *drop-column importance* -mechanism[1]. After training our baseline model with P amount of features, we trained P amount of new models and compared each of the new models' tested accuracy against the baseline model. Should a feature affect the accuracy of the model, the model trained with that feature dropped from the dataset would have a lower accuracy score than the baseline model. The more the accuracy of the model drops with a feature removed, the more important that feature is to the model when classifying pull-requests as accepted or rejected. In table 6.5 we show the importance of the 20 most common quality rules when comparing the baseline model accuracy with a model that has the specific quality rule dropped from the feature set.

Summary of RQ2 and RQ3

Looking at the results we obtained from the analysis using statistical and machine learning techniques (χ^2 0.12 and AUC 50% on average), code quality does not appear to influence pull request acceptance.

[1]https://explained.ai/rf-importance/

7 DISCUSSION

In this Section, we will discuss the results obtained according to the RQs and present possible practical implications from our research.

The analysis of the pull requests in 28 well-known Java projects shows that code quality, calculated by means of PMD rules, is not a driver for the acceptance or the rejection of pull requests. PMD recommends manual customization of the set of rules instead of using the out-of-the-box rule set and selecting the rules that developers should consider in order to maintain a certain level of quality. However, since we analyzed all the rules detected by PMD, no rule would be helpful and any customization would be useless in terms of being able to predict the software quality in code submitted to a pull request. The result cannot be generalized to all the open source and commercial projects, as we expect some project could enforce quality checks to accept pull requests. Some tools, such as SonarQube (one of the main PMD competitor), recently launched a new feature to allow developers to check the TD Issues before submitting the pull requests. Even if maintainers are not sensible to the quality of the code to be integrated in their projects, at least based on the rules detected by PMD, the adoption of pull request quality analysis tools such as SonarQube or the usage of PMD before submitting a pull request will increase the quality of their code, increasing the overall software maintainability and decreasing the fault proneness that could be increased from the injection of some TD Items (see Table I).

The results complement those obtained by Soares et al. [65] and Calefato et al. [16], namely, that the reputation of the developer might be more important than the quality of the code developed. The main implication for practitioners, and especially for those maintaining open-source projects, is the realization that they should pay more attention to software quality. Pull requests are a very powerful instrument, which could provide great benefits if they were used for code reviews as well. Researchers should also investigate whether other quality aspects might influence the acceptance of pull requests.

8 THREATS TO VALIDITY

In this Section, we will introduce the threats to validity following the structure suggested by Yin [71] and discussing construct validity, internal validity, external validity, and conclusion validity. Moreover, we will also present the different tactics adopted to mitigate them.

Construct Validity. This threat concerns the relationship between theory and observation due to possible measurement erorrs. Above all, we relied on PMD, one of the most used software quality analysis tool for Java. However, beside PMD is largely used in industry, we did not find any evidence or empirical study assessing its detection accuracy. Therefore, we cannot exclude the presence of false positive and false negative in the detected TD Items. We extracted the code submitted in pull requests by means of the GitHub API[10]. However, we identified whether a pull request was accepted or not by checking whether the pull request had been marked as merged into the master branch or whether the pull request had been closed by an event that committed the changes to the master branch. Other ways of handling pull requests within a project were not considered and, therefore, we are aware that there could be the limited possibility that some maintainer could have integrated the pull request code into their projects manually, without marking the pull request as accepted. Also, we found a few pull requests that contained sweeping refactoring to the whole project folder layout, resulting in Git marking the majority of the codebase as changed when in reality no new functionality was introduced. This could have resulted in increased number of false positive code quality issues found in the projects.

Internal Validity. This threat concerns internal factors related to the study that might have affected the results. In order to evaluate the code quality of pull requests, we applied the rules provided by PMD, which is one of the most widely used static code analysis tools for Java on the market, also considering the different severity levels of each rule provided by PMD. We are aware that the presence or the absence of a PMD issue cannot be the perfect predictor for software quality, and other rules or metrics detected by other tools could have brought to different results.

External Validity. This threat concerns the generalizability of the results. We selected 28 projects. 21 of them were from the Apache Software Foundation, which incubates only certain systems that follow specific and strict quality rules. The remaining six projects were selected with the help of the trending Java repositories list provided by GitHub. In the selection, we preferred projects that are considered ready for production environ-

ments and are using pull requests as the main way of taking in contributions. Our case study was not based only on one application domain. This was avoided since we aimed to find general mathematical models for the prediction of the number of bugs in a system. Choosing only one domain or a very small number of application domains could have been an indication of the non-generality of our study, as only prediction models from the selected application domain would have been chosen. The selected projects stem from a very large set of application domains, ranging from external libraries, frameworks, and web utilities to large computational infrastructures. The application domain was not an important criterion for the selection of the projects to be analyzed, but at any rate we tried to balance the selection and pick systems from as many contexts as possible. However, we are aware that other projects could have enforced different quality standards, and could use different quality check before accepting pull requests. Furthermore, we are considering only open source projects, and we cannot speculate on industrial projects, as different companies could have different internal practices. Moreover, we also considered only Java projects. The replication of this work on different languages and different projects may bring to different results.

Conclusion Validity. This threat concerns the relationship between the treatment and the outcome. In our case, this threat could be represented by the analysis method applied in our study. We reported the results considering descriptive statistics. Moreover, instead of using only Logistic Regression, we compared the prediction power of different classifier to reduce the bias of the low prediction power that one single classifier could have. We do not exclude the possibility that other statistical or machine learning approaches such as Deep Learning or others might have yielded similar or even better accuracy than our modeling approach. However, considering the extremely low importance of each TD Issue and its statistical significance, we do not expect to find big differences applying other type of classifiers.

9 CONCLUSION

In this study, we looked at the background of pull requests, code quality and machine learning. Then we looked at the previous works on the matter and they reported that 84% of pull requests to be accepted based on the trustworthiness of the developers [31][16]. However, no works studying the effects of code quality on pull requests were found. We believed that the maintainers of open-source projects would consider code quality as a factor when deciding whether to accept of reject an incoming pull request to their project.

In order to verify this belief, a script to gather data and analyze it was created. The code quality of pull requests in 28 different open-source projects were analyzed with the help of PMD, which is one of the most widely used static code analysis tools available. PMD can detect different types of flaws in the source code, including design flaws, code smells, security vulnerabilities, patterns that can lead to potential bugs emerging and many other issues. PMD was able to detect a good number of code quality issues in the analyzed pull requests that have been empirically considered as harmful by several works. Of the 36,344 pull requests analyzed nearly half had been accepted and the other half rejected. Nearly all of the different code quality issues types encountered were present in both accepted and rejected pull requests.

By applying basic statistical techniques and eight machine learning classifiers, we created models that could take in previously unseen examples of code quality measurements and predict whether the pull request would be accepted or rejected by the maintainers of the project. The accuracy measures from the created models indicate that contrary to our assumption, code quality issues present in the pull request's source code do not affect the rate of acceptance by the maintainers.

Results complement the conclusions derived by Gausios et al. [31] and Calefato et al. [16], who report that the reputation of the developer submitting the pull request is one of the most important acceptance factors.

Future works include the replication of this work on a larger dataset, such as [42] and the comparison of our results with other works such as [60].

REFERENCES

[1] A. F. Ackerman, L. S. Buchwald and F. H. Lewski. Software inspections: an effective verification process. *IEEE Software* 6.3 (May 1989), 31–36. ISSN: 0740-7459. DOI: 10.1109/52.28121.

[2] A. F. Ackerman, P. J. Fowler and R. G. Ebenau. Software Inspections and the Industrial Production of Software. *Proc. Of a Symposium on Software Validation: Inspection-testing-verification-alternatives*. Darmstadt, Germany, 1984, 13–40.

[3] J. Al Dallal and A. Abdin. Empirical Evaluation of the Impact of Object-Oriented Code Refactoring on Quality Attributes: A Systematic Literature Review. *IEEE Transactions on Software Engineering* 44.1 (2018), 44–69.

[4] H. Aman. An Empirical Analysis on Fault-Proneness of Well-Commented Modules. *2012 Fourth International Workshop on Empirical Software Engineering in Practice*. Oct. 2012, 3–9.

[5] H. Aman, S. Amasaki, T. Sasaki and M. Kawahara. Empirical Analysis of Fault-Proneness in Methods by Focusing on their Comment Lines. *2014 21st Asia-Pacific Software Engineering Conference*. Vol. 2. Dec. 2014, 51–56.

[6] A. Bacchelli and C. Bird. Expectations, Outcomes, and Challenges of Modern Code Review. *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA, 2013, 712–721. ISBN: 978-1-4673-3076-3.

[7] V. R. Basili, G. Caldiera and H. D. Rombach. The Goal Question Metric Approach. *Encyclopedia of Software Engineering* (1994).

[8] M. Beller, R. Bholanath, S. McIntosh and A. Zaidman. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. *23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. Mar. 2016, 470–481.

[9] A. P. Bradley. The use of the area under the ROC curve in the evaluation of machine learning algorithms. *Pattern Recognition* 30.7 (1997), 1145–1159.

[10] L. Breiman, J. Friedman, C. Stone and R. Olshen. *Classification and Regression Trees*. The Wadsworth and Brooks-Cole statistics-probability series. Taylor & Francis, 1984.

[11] L. Breiman. Bagging Predictors. *Machine Learning* 24.2 (Aug. 1996), 123–140.

[12] L. Breiman. *Arcing the edge*. Tech. rep. Technical Report 486, Statistics Department, University of California at . . ., 1997.

[13] L. Breiman. Random Forests. *Machine Learning* 45.1 (Oct. 2001), 5–32.

[14] W. H. Brown, R. C. Malveau, H. W. " McCormick and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. 1st. New York, NY, USA, 1998. ISBN: 0471197130, 9780471197133.

[15] W. J. Brown, R. C. Malveau, H. W. " McCormick and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis: Refactoring Software, Architecture and Projects in Crisis*. John Wiley and Sons, 1998. ISBN: 0471197130.

[16] F. Calefato, F. Lanubile and N. Novielli. A Preliminary Analysis on the Effects of Propensity to Trust in Distributed Software Development. *2017 IEEE 12th International Conference on Global Software Engineering (ICGSE)*. May 2017, 56–60. DOI: 10.1109/ICGSE.2017.1.

[17] T. Chen and C. Guestrin. XGBoost: A Scalable Tree Boosting System. *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2016, 785–794.

[18] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Trans. Softw. Eng.* 20.6 (June 1994), 476–493. ISSN: 0098-5589.

[19] D. R. Cox. The Regression Analysis of Binary Sequences. *Journal of the Royal Statistical Society. Series B (Methodological)* 20.2 (1958), 215–242.

[20] W. Cunningham. The WyCash Portfolio Management System. OOPSLA '92. 1992. ISBN: 0-89791-610-7.

[21] M. D'Ambros, A. Bacchelli and M. Lanza. On the Impact of Design Flaws on Software Defects. *2010 10th International Conference on Quality Software*. July 2010, 23–31.

[22] B. de Alwis and J. Sillito. Why are software projects moving from centralized to decentralized version control systems?: *2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*. May 2009, 36–39. DOI: 10.1109/CHASE.2009.5071408.

[23] B. Du Bois, S. Demeyer, J. Verelst, T. Mens and M. Temmerman. Does God Class Decomposition Affect Comprehensibility?: Jan. 2006, 346–355.

[24] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal* 15.3 (1976), 182–211. ISSN: 0018-8670. DOI: 10.1147/sj.153.0182.

[25] D. G. Feitelson, E. Frachtenberg and K. L. Beck. Development and Deployment at Facebook. *IEEE Internet Computing* 17.4 (July 2013), 8–17. ISSN: 1089-7801. DOI: 10.1109/MIC.2013.25.

[26] F. Fontana Arcelli and S. Spinelli. Impact of Refactoring on Quality Code Evaluation. *Proceedings of the 4th Workshop on Refactoring Tools*. WRT '11. Waikiki, Honolulu, HI, USA, 2011, 37–40. ISBN: 978-1-4503-0579-2.

[27] M. Fowler and K. Beck. Refactoring: Improving the Design of Existing Code. *Addison-Wesley Longman Publishing Co., Inc.* (1999).

[28] Y. Freund and R. E. Schapire. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *Journal of Computer and System Sciences* 55.1 (1997), 119–139.

[29] J. H. Friedman. Greedy function approximation: A gradient boosting machine. *Ann. Statist.* 29.5 (Oct. 2001), 1189–1232.

[30] P. Geurts, D. Ernst and L. Wehenkel. Extremely randomized trees. *Machine Learning* 63.1 (Apr. 2006), 3–42.

[31] G. Gousios, A. Zaidman, M. Storey and A. van Deursen. Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective. *37th IEEE International Conference on Software Engineering*. Vol. 1. May 2015, 358–368.

[32] G. Gousios, M. Pinzger and A. van Deursen. An Exploratory Study of the Pull-based Software Development Model. *36th International Conference on Software Engineering*. ICSE 2014. 2014, 345–355.

[33] V. J. Hellendoorn, P. T. Devanbu and A. Bacchelli. Will They Like This? Evaluating Code Contributions with Language Models. *12th Working Conference on Mining Software Repositories*. May 2015, 157–167.

[34] *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. Standard. Geneva, CH: International Organization for Standardization, Mar. 2011.

[35] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, F. Khomh and M. Zulkernine. Evaluating the Impact of Design Pattern and Anti-pattern Dependencies on Changes and Faults. *Empirical Softw. Engg.* 21.3 (June 2016), 896–931.

[36] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German and D. Damian. An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering* 21.5 (Oct. 2016), 2035–2071.

[37] F. Khomh, M. Di Penta and Y. Gueheneuc. An Exploratory Study of the Impact of Code Smells on Software Change-proneness. *2009 16th Working Conference on Reverse Engineering*. Oct. 2009, 75–84.

[38] F. Khomh, M. Di Penta and Y. Gueheneuc. An Exploratory Study of the Impact of Code Smells on Software Change-proneness. *2009 16th Working Conference on Reverse Engineering*. Oct. 2009, 75–84.

[39] O. Kononenko, T. Rose, O. Baysal, M. Godfrey, D. Theisen and B. de Water. Studying Pull Request Merges: A Case Study of Shopify's Active Merchant. *40th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP '18. 2018, 124–133.

[40] M. Lanza, R. Marinescu and S. Ducasse. *Object-Oriented Metrics in Practice*. Berlin, Heidelberg: Springer-Verlag, 2005. ISBN: 3540244298.

[41] D. Legay, A. Decan and T. Mens. On the impact of pull request decisions on future contributions. *CoRR* abs/1812.06269 (2018). arXiv: `1812.06269`. URL: `http://arxiv.org/abs/1812.06269`.

[42] V. Lenarduzzi, N. Saarimäki and D. Taibi. The Technical Debt Dataset. *Int. Conf. on Predictive Models and Data Analytics in software engineering (PROMISE'19)*. Sept. 2019.

[43] V. Lenarduzzi, A. Sillitti and D. Taibi. A Survey on Code Analysis Tools for Software Maintenance Prediction. *Software Engineering for Defence Applications — SEDA*. 2019.

[44] W. Li and R. Shatnawi. An Empirical Study of the Bad Smells and Class Error Probability in the Post-release Object-oriented System Evolution. *J. Syst. Softw.* 80.7 (July 2007), 1120–1128.

[45] T. J. McCabe. A Complexity Measure. *IEEE Trans. Softw. Eng.* 2.4 (July 1976), 308–320. ISSN: 0098-5589.

[46] T. Mitchell. *Machine Learning*. McGraw-Hill International Editions. McGraw-Hill, 1997. ISBN: 9780071154673. URL: https://books.google.fi/books?id=EoYBngEACAAJ.

[47] M. Mohri, A. Rostamizadeh and A. Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2012. ISBN: 026201825X, 9780262018258.

[48] M. Nagappan, T. Zimmermann and C. Bird. Diversity in Software Engineering Research. ESEC/FSE 2013. Saint Petersburg, Russia, 2013, 466–476.

[49] S. M. Olbrich, D. S. Cruzes and D. I. K. Sjøberg. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. *2010 IEEE International Conference on Software Maintenance*. Sept. 2010, 1–10. DOI: 10.1109/ICSM.2010.5609564.

[50] S. Olbrich, D. S. Cruzes, V. Basili and N. Zazworka. The evolution and impact of code smells: A case study of two open source systems. *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. Oct. 2009, 390–400. DOI: 10.1109/ESEM.2009.5314231.

[51] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto and A. D. Lucia. On the Diffuseness and the Impact on Maintainability of Code Smells: A Large Scale Empirical Investigation. *Empirical Softw. Engg.* 23.3 (June 2018), 1188–1221.

[52] M. Patton. *Qualitative Evaluation and Research Methods*. Newbury Park: Sage, 2002.

[53] R. Potvin and J. Levenberg. Why Google Stores Billions of Lines of Code in a Single Repository. *Commun. ACM* 59.7 (June 2016), 78–87.

[54] D. Powers. Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness Correlation. *Mach. Learn. Technol.* 2 (Jan. 2008).

[55] M. M. Rahman, C. K. Roy and J. A. Collins. CORRECT: Code Reviewer Recommendation in GitHub Based on Cross-Project and Technology Experience. *38th International Conference on Software Engineering Companion (ICSE-C)*. May 2016, 222–231.

[56] M. M. Rahman and C. K. Roy. An Insight into the Pull Requests of GitHub. *11th Working Conference on Mining Software Repositories*. MSR 2014. 2014, 364–367.

[57] P. C. Rigby and M. Storey. Understanding broadcast based peer review on open source software projects. *33rd International Conference on Software Engineering (ICSE)*. May 2011, 541–550.

[58] P. Rigby, B. Cleary, F. Painchaud, M. Storey and D. German. Contemporary Peer Review in Action: Lessons from Open Source Development. *IEEE Software* 29.6 (Nov. 2012), 56–61. ISSN: 0740-7459. DOI: 10.1109/MS.2012.24.

[59] P. Runeson and M. Höst. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Softw. Engg.* 14.2 (2009), 131–164.

[60] N. Saarimäki, V. Lenarduzzi and D. Taibi. On the diffuseness of code technical debt in open source projects of the Apache Ecosystem. *International Conference on Technical Debt (TechDebt 2019)* (2019).

[61] R. E. Schapire. The strength of weak learnability. *Machine Learning* 5.2 (June 1990), 197–227.

[62] J. Schumacher, N. Zazworka, F. Shull, C. Seaman and M. Shaw. Building Empirical Support for Automated Code Smell Detection. *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '10. Bolzano-Bozen, Italy, 2010, 8:1–8:10.

[63] F. Shull and C. Seaman. Inspecting the History of Inspections: An Example of Evidence-Based Technology Diffusion. *IEEE Software* 25.1 (Jan. 2008), 88–90. ISSN: 0740-7459. DOI: 10.1109/MS.2008.7.

[64] D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus and T. Dybå. Quantifying the Effect of Code Smells on Maintenance Effort. *IEEE Transactions on Software Engineering* 39.8 (Aug. 2013), 1144–1156.

[65] D. M. Soares, M. L. d. L. Júnior, L. Murta and A. Plastino. Rejection Factors of Pull Requests Filed by Core Team Developers in Software Projects with High Acceptance Rates. *14th International Conference on Machine Learning and Applications (ICMLA)*. 2015, 960–965.

[66] D. M. Soares, M. L. de Lima Júnior, A. Plastino and L. Murta. What factors influence the reviewer assignment to pull requests?: *Information and Software Technology* 98 (2018), 32–43. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2018.01.015. URL: http://www.sciencedirect.com/science/article/pii/S0950584917303804.

[67] J. Tsay, L. Dabbish and J. Herbsleb. Influence of Social and Technical Factors for Evaluating Contribution in GitHub. *36th International Conference on Software Engineering*. ICSE 2014. 2014, 356–366.

[68] E. v. d. Veen, G. Gousios and A. Zaidman. Automatically Prioritizing Pull Requests. *12th Working Conference on Mining Software Repositories*. May 2015, 357–361.

[69] B. A. Wichmann, A. A. Canning, D. L. Clutterbuck, L. A. Winsborrow, N. J. Ward and D. W. R. Marsh. Industrial perspective on static analysis. *Software Engineering Journal* 10.2 (Mar. 1995), 69–75. ISSN: 0268-6961. DOI: 10.1049/sej.1995.0010.

[70] A. Yamashita. Assessing the Capability of Code Smells to Explain Maintenance Problems: An Empirical Study Combining Quantitative and Qualitative Data. *Empirical Softw. Engg.* 19.4 (Aug. 2014), 1111–1143.

[71] R. Yin. *Case Study Research: Design and Methods, 4th Edition (Applied Social Research Methods, Vol. 5)*. 4th. SAGE Publications, Inc, 2009.

[72] Y. Yu, H. Wang, V. Filkov, P. Devanbu and B. Vasilescu. Wait for It: Determinants of Pull Request Evaluation Latency on GitHub. *12th Working Conference on Mining Software Repositories*. May 2015, 367–371.

[73] Y. Yu, H. Wang, G. Yin and C. X. Ling. Reviewer Recommender of Pull-Requests in GitHub. *IEEE International Conference on Software Maintenance and Evolution*. Sept. 2014, 609–612.

[74] F. Zampetti, L. Ponzanelli, G. Bavota, A. Mocci, M. D. Penta and M. Lanza. How Developers Document Pull Requests with External References. *25th International Conference on Program Comprehension (ICPC)*. Vol. 00. May 2017, 23–33.

[75] F. Zampetti, G. Bavota, G. Canfora and M. Di Penta. A Study on the Interplay between Pull Request Review and Continuous Integration Builds. Feb. 2019, 38–48.

[76] N. Zazworka, M. A. Shaw, F. Shull and C. Seaman. Investigating the Impact of Design Debt on Software Quality. *Proceedings of the 2Nd Workshop on Managing Technical Debt*. MTD '11. Waikiki, Honolulu, HI, USA, 2011, 17–23.

[77] X. Zhang, Y. Chen, Y. Gu, W. Zou, X. Xie, X. Jia and J. Xuan. How do Multiple Pull Requests Change the Same Code: A Study of Competing Pull Requests in GitHub. *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Sept. 2018, 228–239. DOI: 10.1109/ICSME.2018.00032.