MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Source Code Quality impact on Pull Requests acceptance

Master's Thesis

ONDŘEJ KUHEJDA

Advisor: Assistant professor Bruno Rossi

Department of Computer Systems and Communications

Brno, Spring 2022

MUNI
FI

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Ondřej Kuhejda

Advisor: Assistant professor Bruno Rossi

Acknowledgements

I would like to thank my supervisor, Bruno Rossi, for his guidance throughout the whole process.

Computational resources were supplied by the project "e-Infrastruktura CZ" supported by the Ministry of Education, Youth and Sports of the Czech Republic.

Abstract

TODO

Keywords

code quality, pull request, static code analysis

Contents

1 Introduction

1.1 Problem statement

RQ$_1$ Which code issues are typically introduced by the pull requests?

RQ$_2$ Are there some particular issues/code smells that affect the pull request acceptance?

RQ$_3$ Is there a relationship between the source code quality and the pull request acceptance?

RQ$_4$ Does code quality influence the time it takes to close a pull request?

RQ$_5$ Is code quality impact higher in projects that are using some particular programming language?

2 Code quality in pull-based development

The pull-based development model created novel ways how can developers interact between each other. Instead of pushing code changes (patches) into one central repository, developers can work in more decentralized and distributed way. This is mainly done by using distributed version control systems such as Git. Git enables developers to clone repositories and thus to work independently on projects. Furthermore, the Git's branching model helps developers to keep track of repository changes and helps to handle the conflicts between the different changes of the same code base.

To furthermore ease the complicated process of resolving conflicts between different changes (of the same code base) and to provide a more user-friendly environment for developers, platforms such as GitHub was created. These platforms adds new ways how the developers can interact beyond the basic functionality of Git:

- The forks enables to create the server-side copy of the repository.
- Pull requests (on some platforms called merge requests) enables to merge code directly on the platform.
- Users can report issues found in the projects; therefore, platform can also serve as a bug-tracking system.
- The comments can be added to the pull requests and issues in order to build up social interaction between developers.
- Users can star projects and follow other users, projects, pull requests or issues.

In this study, I choose to use GitHub as the main source for data mining. GitHub is one of the leading platforms that enables pull-based collaboration between developers. GitHub hosts huge amount of publicly available repositories and GitHub also provides public REST API that can be easily leveraged for data mining.

The aim of this thesis is to obtain large amount of data about GitHub projects and analyze the pull request in regard of their code quality. How the code quality can be analyzed and how the GitHub

platforms contributes to quality of the code itself is discussed in the following chapters.

- **TODO** cite: An Exploratory Study of the Pull-based Software Development Model

2.1 Code quality

Code quality is very important aspect of every program — software with high code quality has competitive advantage, is more stable and is also more maintainable then software which is poorly written.

To be able to evaluate the software in regard of its quality, there needs to be some way how can be code quality measured. The testing can be used exactly for this purpose — as a tool for measuring the quality of the source code. There are multiple ways how can be testing performed. Testing techniques can be divided into two categories: static and dynamic testing techniques.

In order to use dynamic testing techniques on large number of programs, there are two large obstacles — the program needs to be executed and there needs to be some inputs (with expected outputs) that can be then used for testing. Program execution can be problematic. Some programs needs to be compiled before they can be executed; others requires special environment for its execution (specific hardware, operating system or shared libraries required by the program). Moreover, the most of the programs does not have sets of input that can be used for testing. There exists some techniques that can be used also without the predefined inputs such as fuzzing, but these techniques are usually time-consuming. Because of that, dynamic testing techniques are not viable option when dealing with the large number of programs.

On the other hand, static testing methods suits the analysis of the large number of programs better. Static techniques include usage formal and informal reviews, walkthroughs and inspections; however, these techniques are performed by humans and therefore are not usable for large datasets. Because of that, in this thesis, the quality of the given source code is evaluated using the tools for automatic static analysis (called linters). Linters are used to find defects and code smells in the source code without the need of source code's execution.

There are several categories of issues which can be detected using linters. Source code can be checked if it follows a conventions of the given programming language. For instance, Python has an official style guide for Python code — PEP 8[1]. This guide defines the conventions that should be followed such as proper indentation of the code blocks, maximum line length or naming conventions.

Furthermore, code can be analyzed against refactoring related checks; for instance linter can detect if some part of the code is redundant and therefore could be omitted. Linters can also detect actual errors such as type mismatches or syntax errors.

However, it is important to note that not all linters have the same capabilities. Number of issues which can be detected by the given linter also heavily depends on the programming language of the studied source code. Which linters were used for the purposes of this thesis is discussed later in the text.

- **TODO** cite: `https://www.utcluj.ro/media/page_document/78/Foundations%20of%20software%20testing%20-%20ISTQB%20Certification.pdf`

2.2 TODO GitHub

- GitHub issues and code quality
- Ways to merge code
  - An Exploratory Study of the Pull-based Software Development Model
- PRs and code review
- PRs CI/CD and code quality
  - Wait for It: Determinants of Pull Request Evaluation Latency on GitHub [1]
    * CI and latency

1. `https://www.python.org/dev/peps/pep-0008/`

- Trautsch et al. [2] analyzed several open-source projects in regards to usage of static analysis tools. They found out that incorporating a static analysis tool in a build process reduces the defect density.

3 Pull request acceptance

Pull request acceptance is a problem that has been studied multiple times. Several surveys were performed in order to understand why pull requests are being rejected.

Gousios et al. [3] surveyed hundreds of integrators to find out their reasons behind the PR rejection. Code quality was stated as the main reason by most of the integrators; code style was in the second place. Factors that integrators examine the most when evaluating the code quality are style conformance and test coverage.

Kononenko et al. [4] performed a study of an open-source project called *Shopify*; they manually analyzed PR's and also surveyed *Shopify* developers. They found out that developers associate the quality of PR with the quality of its description and with the revertability and complexity of the PR.

The reasons why contributors abandon their PRs were also studied [5]. The reason number one was the "Lack of answers from integrators."; moreover, the "Lack of time" and the "Pull request is obsolete" was also often stated as the main reason.

Even though the different open-source communities solve the problem of pull request acceptance in a different manner, three main governance styles can be identified — protective, equitable, lenient. Protective governance style values trust in the contributor-maintainer relationship. The equitable governance style tries to be unbiased towards the contributors, and the lenient style prioritizes the growth and openness of the community [6]. Each style focuses on different aspects of PR. Tsay et al. [7] identified the following levels of social and technical factors that influence the acceptance of the PR — *repository level*, *submitter level*, and the *pull request level*.

3.1 Repository level

The *repository level* is interested in the aspects of the repository itself, such as the repository age, number of collaborators, or number of stars on the GitHub.

For instance, the programming language used in the project also influences the acceptance of the PRs. Pull requests containing Java,

JavaScript, or C++ code have a smaller chance to be accepted than PRs containing the code written in Go or Scala [8].

Furthermore, older projects and projects with a large team have a significantly lower acceptance rate [7].

The popularity of the project also influences the acceptance rate — projects with more stars have more rejected PRs [7].

3.2 Submitter level

The *submitter level* is concerned about the submitter's status in the general community and his status in the project itself. There are several parameters that can be considered when evaluating the submitter's status.

PRs of submitters with higher social connection to the project have a higher probability of being accepted [7].

Submitter status in the general community plays an important role in PR acceptance. If the submitter is also a project collaborator, the likelihood that the PR will be accepted increases by 63.3% [7].

Moreover, users that contributed to a larger number of projects have a higher chance that their PR will be accepted [9].

The gender of the submitter is another factor that plays a role in PR acceptance. A study showed that woman's PR are accepted more often, but only when they are not identifiable as a woman [10].

Personality traits also influence PR acceptance. The *IBM Watson Personality Insights* were used to obtain the personality traits of the PR submitters by analyzing the user's comments. These traits were then used to study PR acceptance. It has been shown that conscientiousness, neuroticism, and extroversion are traits that have positive effects on PR acceptance. The chance that PR will be accepted is also higher when the submitter and closer have different personalities [11].

3.3 Pull request level

The *pull request level* is interested in the data that are connected to the PR itself. For instance, on the *PR level*, one can study if there is a correlation between PR acceptance and the number of GitHub

comments in the PR. Another parameter that can be used is "Number of Files Changed" or "Number of Commits".

One of the factors that negatively influence the acceptance rate is the already mentioned number of commits in the pull request. The high number of commits decreases the probability of acceptance. On the other hand, PR's with only one commit are exceptions — they have a smaller chance to be accepted than pull requests which contain two commits [9].

Another observation is that more discussed PR's has a smaller chance to be accepted [7]. Another study did not find a large difference between accepted and rejected PR's based on the number of comments but found that discussions in rejected PR's have a longer duration [12].

Proper testing is the crucial part of every project, and therefore it also influences the pull request acceptance. PR's including more tests have a higher chance to be accepted, and an increasing number of changed lines decreases the likelihood of PR acceptance [7].

Testing plays a significant role in discovering bugs and therefore leads to higher code quality. On the other hand, many test cases do not have to mean that code has a high quality. The code quality is an essential factor on the *pull request level*, therefore, is this study's main interest. Works that are also interested in the code quality and the pull request acceptance are examined in the following chapter.

Another factor that is closely tied to code quality is the code style. This factor has a small (but not negligible) negative effect on acceptance. This means that PRs with larger code style inconsistency (with the codebase) have a smaller chance of being accepted [13].

3.4 TODO Code quality

Although most integrators view code quality as the most important factor regarding PR acceptance, to the best of my knowledge, only one study was performed to discover whether there is a connection between the PR's acceptance and its quality.

- Does code quality affect pull request acceptance? [14]

3.5 TODO Unsorted

- study "Influence of Social and Technical Factors" [7] was replicated [11]
- Replication Can Improve Prior Results: A GitHub Study of Pull Request Acceptance [15]
  - contains interesting table with factors that influences acceptance
- Pull Request Decision Explained: An Empirical Overview [16]
  - also contains interesting table with factors that influences acceptance
- An Exploratory Study of the Pull-Based Software Development Model [17]
- Which Pull Requests Get Accepted and Why? A study of popular NPM Packages [18]
- Rejection Factors of Pull Requests Filed by Core Team Developers in Software Projects with High Acceptance Rates [19]
- Pull Request Prioritization Algorithm based on Acceptance and Response Probability [20]

3.6 TODO Create table that compares already performed studies with my thesis

4 Data mining

TODO: update graph Information about the pull requests are retrieved

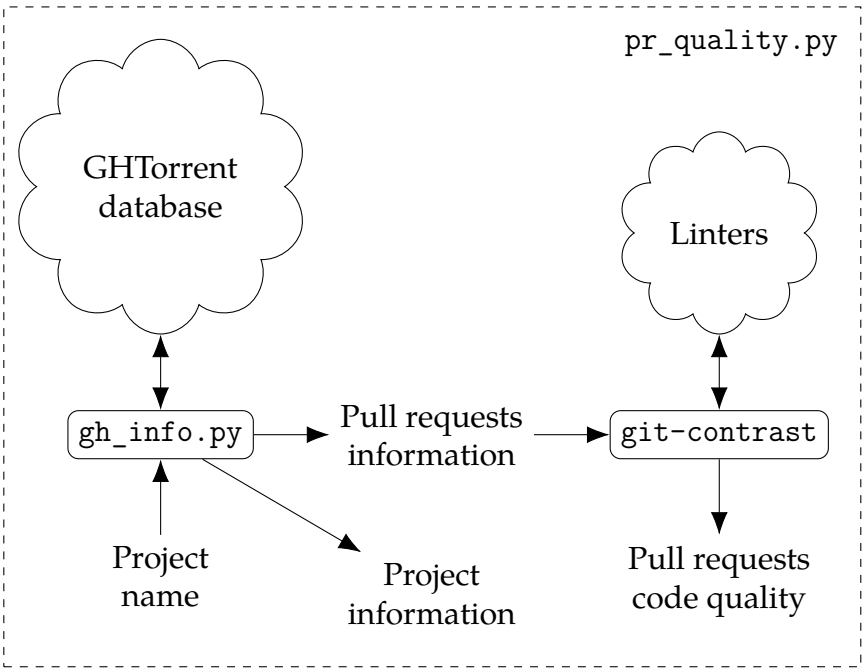

Figure 4.1: The `pr_quality.py` workflow

using the `pr_quality.py` script. This scripts takes names of the projects that will be analyzed as the input and it outputs the JSON files containing the requested data. This script uses internally two other scripts — `gh_info.py` and `git-contrast`.

`gh_info.py` is responsible for querying the GHTorrent database in order to obtain data about the projects. The GHTorrent database is an offline mirror of data offered through the Github REST API. `gh_info.py` returns a JSON file with the information about the project such as number of stars, number of contributors or information about pull requests and their commits.

However, the Github REST API lacks the information about the code quality of the pull requests. This is where the `git-contrast` comes into the play. `git-contrast` is the command-line application which analyzes the code quality of the given pull request using the

external linters. This application is further discussed in the following sections.

TODO: mention that data from REST API are not complete (GH API limit)

4.1 GHTorrent database

As stated before, the script called `gh_info.py` uses the GHTorrent database in order to retrieve GitHub data. GitHub REST API can be leveraged to obtain many interesting factors which can possibly influence the acceptance of pull requests. All the data that are obtained using the `gh_info.py` are listed in the following table:

Table 4.1: Data retrieved from the GHTorrent

Level	Variable	Factor
Repository level	Project name	✗
	Programming language	✓
	Time of creation	✓
	Number of forks	✓
	Number of commits	✓
	Number of project members	✓
	Number of watchers	✓
Submitter level	Username	✗
	Number of followers	✓
	Status in the project	✓
Pull request level	Pull request ID	✗
	Is PR accepted?	✓
	Time opened	✓
	Head repository	✗
	Head commit	✗
	Base commit	✗
	Number of commits	✓
	Number of comments	✓

Variables marked with ticks (✓) are factors that can possibly influence code quality and they can be used for pull request acceptance analysis. Other variables (✗) are not meant to be used as an part of an

data analysis itself, but are kept here for better orientation; and some of them are later used by the `git-contrast` tool (in order to pull the commits which will be subsequently analyzed by linters).

4.2 `git-contrast`

`git-contrast` is the command line application that I implemented in order to be able to analyze the code quality of the given pull request. `git-contrast` expects two commit hashes on the input and returns the information about the change in code quality between these commits on the output. This is done by running the linter on the files in the state of the first commit and then in the state of the second commit. The number of found code quality issues is then written to the standard output.

To measure the change of the quality in the pull request, we simple run the `git-contrast` on the "head commit" and the "base commit" of the given pull request. `git-contrast` supports several linters; which linter will be used is determined by the file extension of the tested file. Linters that are supported by `git-contrast` are listed in the following table:

Table 4.2: Linters supported by the `git-contrast`

Linter	Programming languages	File extensions
OCLint	C/C++	`.c`, `.cpp` and `.h`
HLint	Haskell	`.hs`
ktlint	Kotlin	`.kt` and `.kts`
PMD	Java	`.java`
Pylint	Python	`.py`

The most problematic was to statically analyze the C/C++ source files because some linters also need the information how the source code should be compiled. Luckily, this information can be usually automatically obtained from the makefiles. Another problem is the speed. At first, I was using the Cppcheck linter for the static analysis of C/C++ but I was forced to switch to the OCLint in order to shrink the total execution time of the static analysis.

TODO: add information about versions of linters, issue categories etc.

4.3 TODO Projects selection

Criteria (data from 2019-06-01):

- is in the top 150 most favorite projects written in the given language
- 200+ pull requests and less then 5000
- `https://github.com/EvanLi/Github-Ranking`
- at least 85 % of files are source files written in the given language
- project is a program or program collection (not a book with the script etc.)
- `https://dl.acm.org/doi/abs/10.1145/2597073.2597122`
- `https://dl.acm.org/doi/abs/10.1145/3379597.3387489`
- `https://zenodo.org/record/3858046`
- `https://github.com/XLipcak/rev-rec`
- `https://ghtorrent.org/`
  - `https://github.com/gousiosg/pullreqs`
  - How can I cite this work? (on the web)
- Kalliamvakou et al. noted that data about PR's mined from GitHub are not always reliable, because PR can be also merged using several different approaches.
  - `https://dl.acm.org/doi/10.1145/2597073.2597074`
  - [17]

4.4 TODO Computational resources

5 Data analysis

5.1 TODO Scripts

- Put this into appendix?

Data preprocessing

In order to simplify analysis of retrieved data, I created the script that takes multiple JSON files with the data about each individual project and converts them into the CSV files. Each row in the CSV file represents some pull request. This script also filters the pull requests which are not suitable for the analysis — PRs that do not contains any source code written in the primary language or PRs that contained corrupted files (the linter was unable to analyze those files).

Pull request classification

The retrieved data about the pull request were subsequently analyzed in order to answer my research questions. For this analysis was used the Python script[1] provided by Lenurdazzi et al. (this script was used for machine learning classification methods)

Pull request regression

Main script

- rest of the analysis
- divided by research questions
- aggregation of results from the classification and regression
- export to LaTeX and PDF

1. `https://figshare.com/s/d47b6f238b5c92430dd7`

5.2 Statistical methods

Research questions 1 to 4 were analyzed separately for each programming language.

TODO: For each test, state the rationale behind the test and assumptions about data.

- `https://www.scribbr.com/statistics/statistical-tests/`

Which code issues are typically introduced by the pull requests?

At first, in order to answer the $\mathbf{RQ_1}$, I summarized the retrieved data for each project — I counted how many suitable pull requests were analyzed and how many of them are accepted/rejected. Then I created a scatter plot between number of stars and percentage of accepted PRs.

I also summarized all pull requests regardless of their project. I computed average number of introduced issues, fixed issues etc. Then I created heat map that shows how many PRs introduced/fixed some specific number of issues.

Then for each issue individually I computed how many accepted/rejected pull request introduced/fixed this issue, how many times this issue occurred in some pull request etc. I created multiple lists of issues sorted by some parameter. I sorted issues by the number of rejected/accepted PRs that fixed/introduced them. I also listed issues and percentage of PRs that changed their quality. I examined the issues that were fixed in larger number of PRs then introduced. Then I created a scatter plot that shows which issue category contains the most common issues.

These steps were applied individually for each programming language to determine how does the average PR look line in terms of code quality.

Are there some particular issues/code smells that affect the pull request acceptance?

In order to discover issues that affects the acceptance of pull requests most, the classification models were created. The aim of these models is to classify pull request into two groups (accepted PRs and rejected

PRs) by using the information about quality change in the given pull request. Multiple classification algorithms were used:

LogisticRegression TODO

RandomForest TODO

GradientBoost TODO

ExtraTrees TODO

DecisionTrees TODO

Bagging TODO

AdaBoost TODO

XGBoost TODO

Each of those algorithms was run on three different datasets — dataset with quality change, dataset containing only introduced issues and dataset with only fixed issues. In the first dataset, the quality change for some issue was represented by the integer and this integer was negative if the issue was fixed in the PR and positive if the issues was introduced. The other datasets were created by filtering positive/negative values from the first dataset.

In order to recognize issues that have some effect on the PR acceptance, the *drop-column importance* mechanism[2] was used.

- **TODO:** reference the paper
- **TODO:** correlation matrix
- **TODO:** reference/explain the model reliability (from the appendix)

2. `https://explained.ai/rf-importance/`

Is there a relationship between the source code quality and the pull request acceptance?

RQ_3:

- classification
- PCA scatterplot
- contingency matrices
- contingency matrices for PR's that contain only modified source code files vs for all of them
- contingency matrices separately for each project
  - According to Cochran (1952, 1954), all expected counts should be 10 or greater. If < 10, but $>=5$, Yates' Correction for continuity should be applied.
- ROC curves and AUCs
- mention that p-values were not adjusted

Pull request that are adding or removing some files has large influence on code quality. If the number of removed/added files has large impact on PR acceptance, then it can be a large thread to validity of the independence test. The pull request acceptance can be also influenced by quality of files which were not linted (was written in non-primary language). To eliminate the risk that the test was influenced, same test was performed on pull requests that only modified some source files and these files were written in the primary language.

Does code quality influence the time it takes to close a pull request?

In order to find the possible link between the code quality and time it takes to close a PR, the regression algorithms were used. At first, the dataset was split into two parts — training and test set. After that, the regression model was trained on the training set. Then, the importance of individual quality issues was determined using the *permutation importance* mechanism. To evaluate the model itself, the

model was used to predict the time based on the data from test set. Metrics such as *mean absolute error* (MAE), *mean squared error* (MSE) and *coefficient of determination* (R^2) were computed using the predicted and expected values.

- **TODO:** correlation matrix

Is code quality impact higher in projects that are using some particular programming language?

RQ_5:

- Compare results from previous steps.
  - Statistically compare the parameters obtained for each programming language. (check if the pull requests from different languages and retrieved parameters follow the same distributions)
- What is the effect of the programming language on the acceptance and time it takes to close a pull request?
  - ANOVA
  - classification (machine learning)

6 Evaluation

TODO

6.1 Python

In order to analyze the influence of code quality on the pull request acceptance, the 20 projects from the Python ecosystem were selected. In total, 9452 pull requests were analyzed and 73 % of these PRs was accepted. Pull request were more accepted in less popular projects, as can be seen in the following scatterplot:

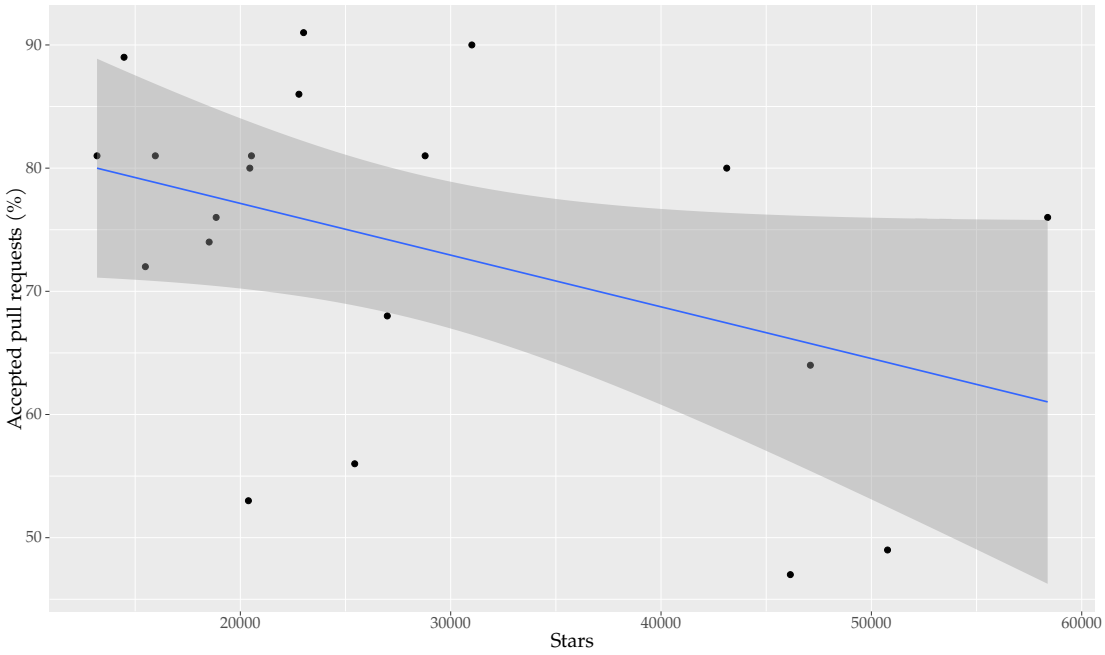

Figure 6.1: Stars and pull request acceptance

At average, one pull request introduced 5.36 issues and fixed 2.44 issues; accepted pull request introduced 4.62 and fixed 1.99 issues, and rejected pull request introduced 7.86 issues and fixed 4.43 on average. 5 % trimmed mean was used to compute these values.

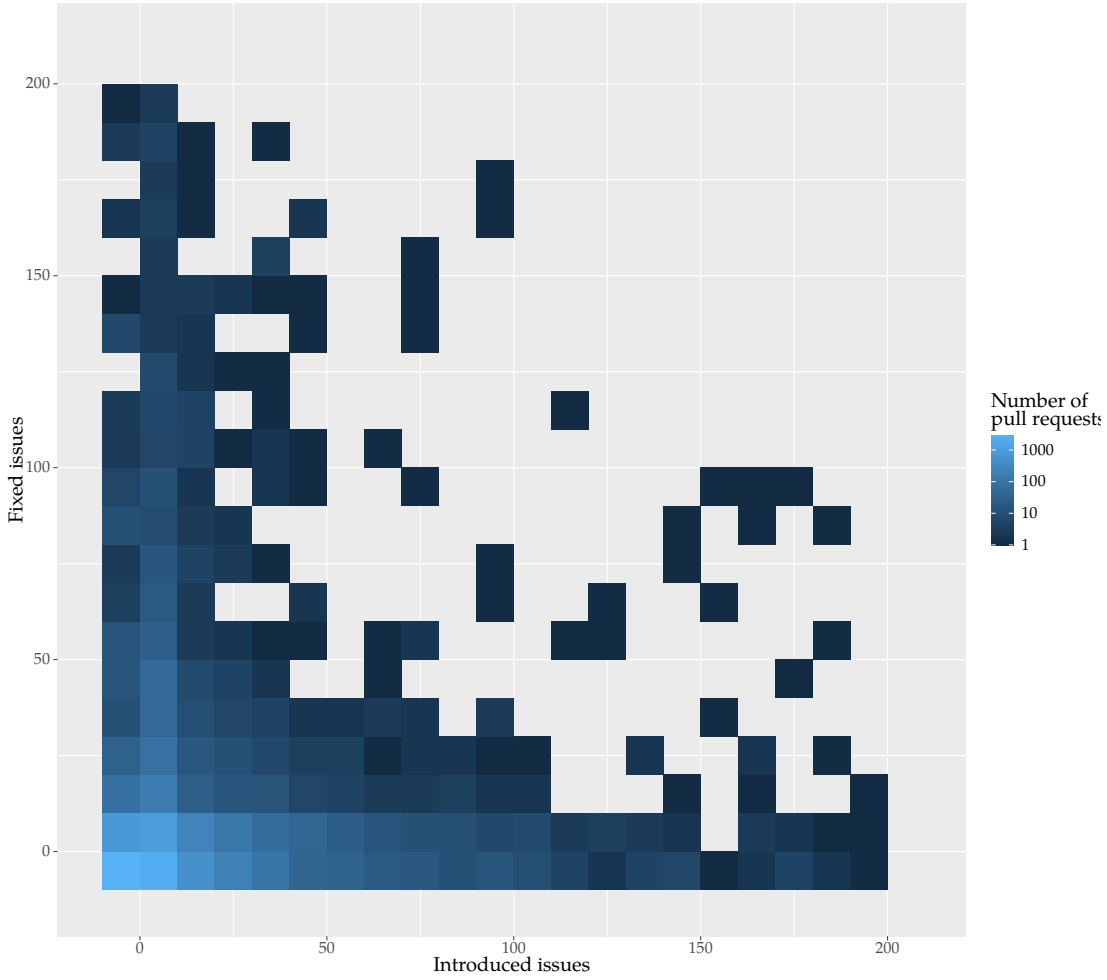

Figure 6.2: Pull requests and quality

In the analyzed pull requests, Pylint detected 222 different issues.

The list of issues that was fixed/introduced in the largest number of pull request was dominated by the conventions. The convention that was fixed/introduced in the largest number of pull requests is `missing-function-docstring` (in 37 % of PRs); also conventions `invalid-name`, `line-too-long` and `consider-using-f-string` were fixed/introduced in over then 20 % of pull requests. There were 15 issues that were fixed/introduced in more then 10 % of PRs and 72 issues were in over 1 % of PRs (out of the 222 issues which were found

in the pull requests). There were 9 issues which was present in the analyzed pull requests but did not influence their quality (number of these issues was not changed by any pull request). 13 issues were introduced/fixed in only one pull request and 10 of them are issues classified as errors. The most common error is `import-error` (24 % of PRs); however, I suspect that there will be a lot of false positives which aroused due to linting in the isolated environment. 60 issues were fixed in more PR's then they were introduced. They are 24 more PRs that fixed the warning `super-init-not-called` than the PR's that introduced it.

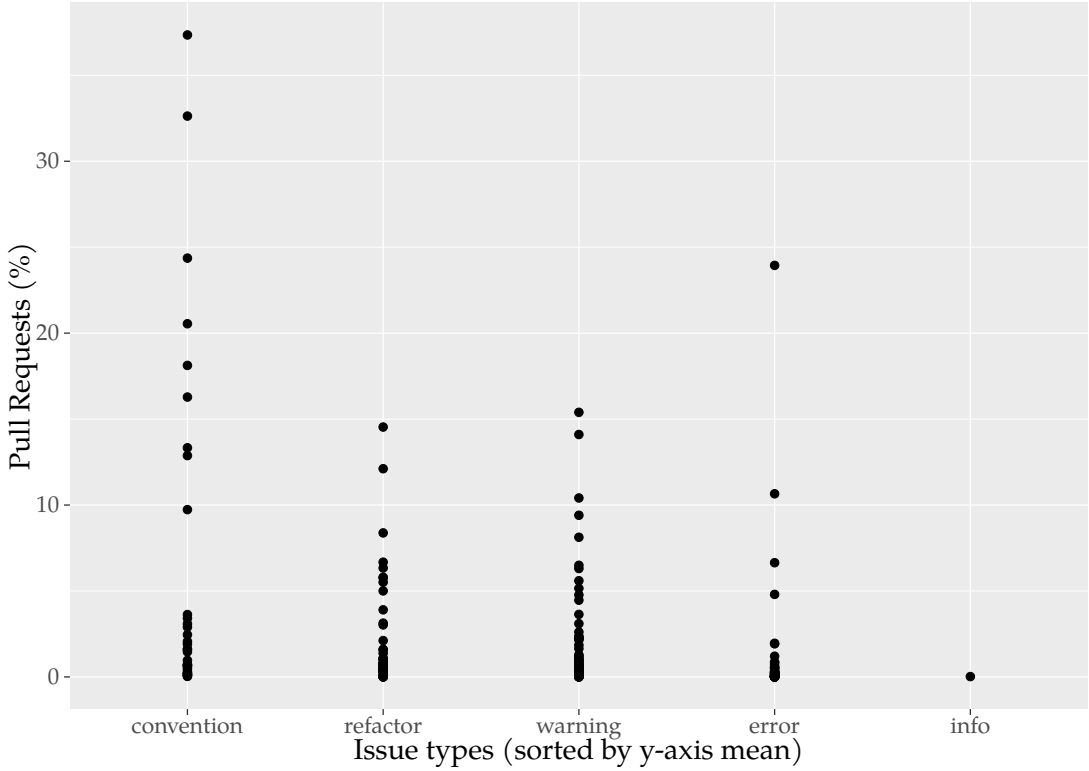

Figure 6.3: Pylint issues and % of PRs which fixed/introduced them

The most important Pylint issue in regards of the PR acceptance is the `syntax-error`. XGBoost classifier gives this error the 1.2 % importance. However, other classifiers consider this error less important. The average importance of the `syntax-error` is only 0.3 %. The syntax

error was introduced in 17 projects. At average, rejected pull request introduced `0.027` syntax errors, and average accepted pull request even fixed `0.001` syntax errors.

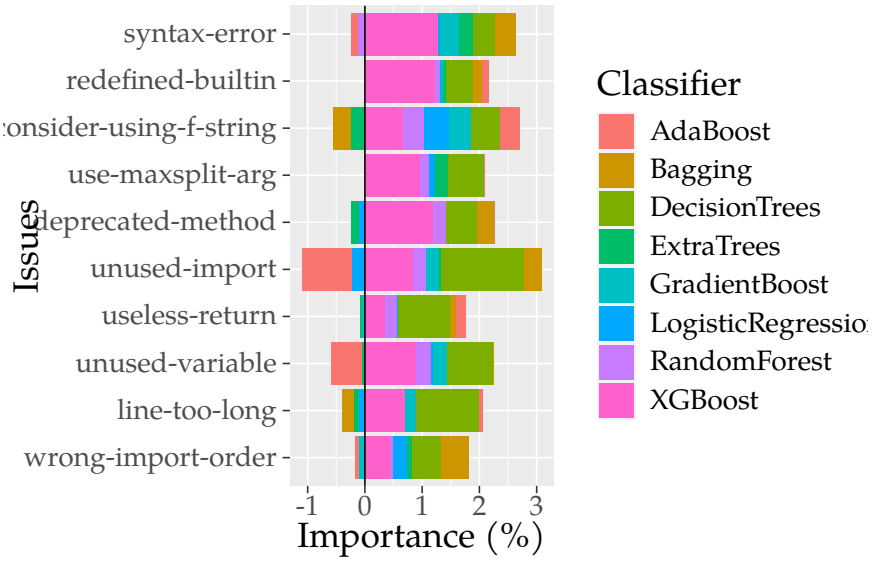

Figure 6.4: Ten most important Pylint issues

When only introduced issues were considered, the list of the most important issues looked differently. On the other hand, there are some issues that appeared in top 10 in both lists: `syntax-error`, `unused-variable` and `unused-import`. The `syntax-error` is considered as the most important issue by both methods.

Using only the information about fixed issues, the most important issue is `f-string-without-interpolation` (in terms of acceptance). However no classifier gives this issue importance over one percent.

In order to visualize the difference in quality between accepted and rejected PRs, I created PCA scatter plot:

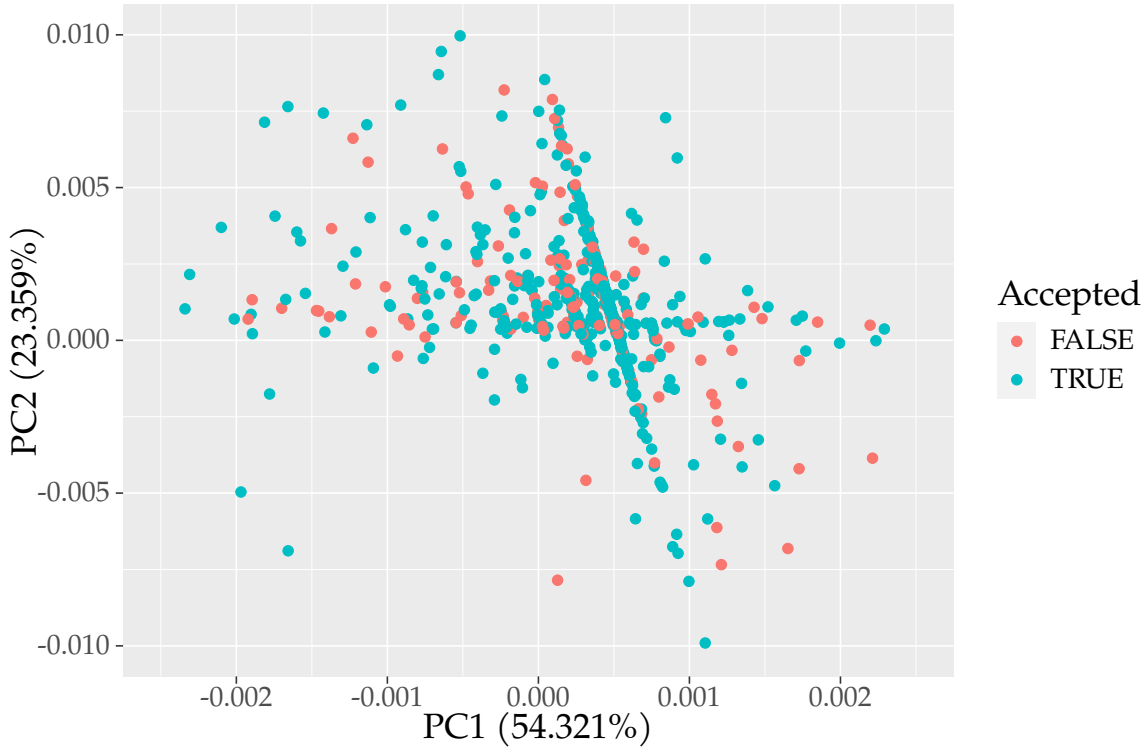

Figure 6.5: PCA scatter plot

In the PCA scatter plot, there is no visible difference between rejected and accepted pull requests.

To understand if the presence of some issue in the PR influences its acceptance, I created contingency matrices and performed chi-square test of independence:

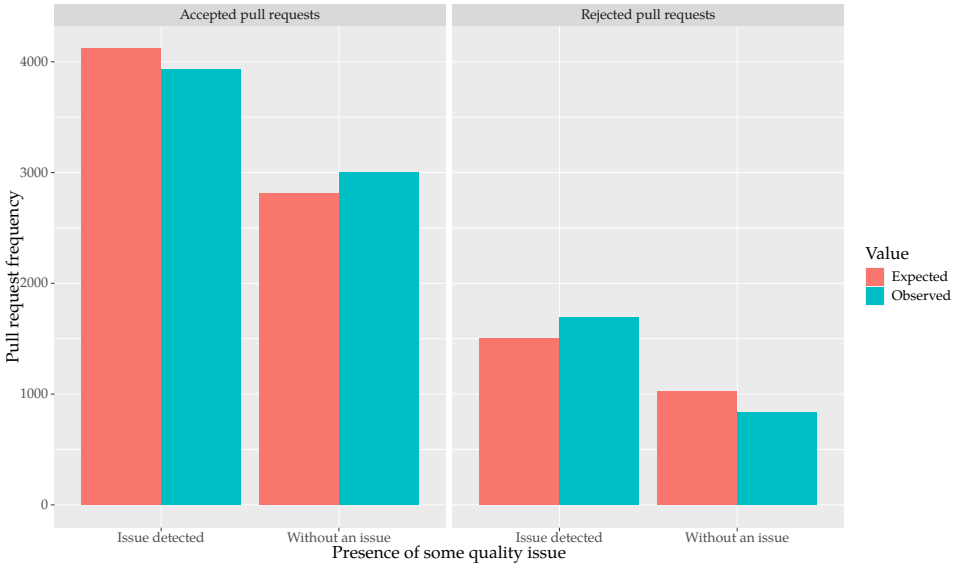

Figure 6.6: Relation between presence of issue and PR acceptance

As can be seen in Figure 6.6, the observed number of rejected pull requests which contained some defected is higher then expected. For chi-square test, $p < 2.2 \times 10^{-16}$ and therefore, the hypothesis that presence of some issue and PR acceptance are independent is rejected on significance level $\alpha = 0.05$. However, the Cramer's $\phi_c \approx 0.092$ therefore the association is weak. This conclusion supports also the fact that AUC for trained classification models is slightly over 50 percent. The average AUC for all models is 53.4.

When considering only PRs that solely modified some source files, $p < 5.548 \times 10^{-10}$ and therefore also there the presence of some code quality issue in the PR influences the PR acceptance. Similarly to the previous test, the $\phi_c \approx 0.087$, therefore the association between presence of same issue and PR acceptance is weak.

Similar results were obtained when the chi-square test was performed separately for each issue category.

When the projects were considered individually, only for nine of them the $p < \alpha$. In these projects, the poor code quality had a negative impact on PR acceptance. In the rest of the projects, presence of some code quality issue does not seems to have an effect on the PR acceptance.

The quality of the code does not seem to have an effect on the time it takes to close a pull request. All of the trained regression models have negative R^2 score (when evaluated on the test set). This means that trained models are worse at predicting the time than a constant (mean value). Similar results were obtained when only introduced issues were considered and also when only fixed issues were considered.

6.2 Java

The next programming language that was analyzed is Java. In total, the 8887 pull requests were linted and the 73 % of these pull request were accepted. At average, the one pull request introduced 20 new PMD issues, but at the same time, also fixed 18 other issues.

Similarly to the Python projects, the pull request from the less popular project were more likely to be accepted than pull request from more popular projects.

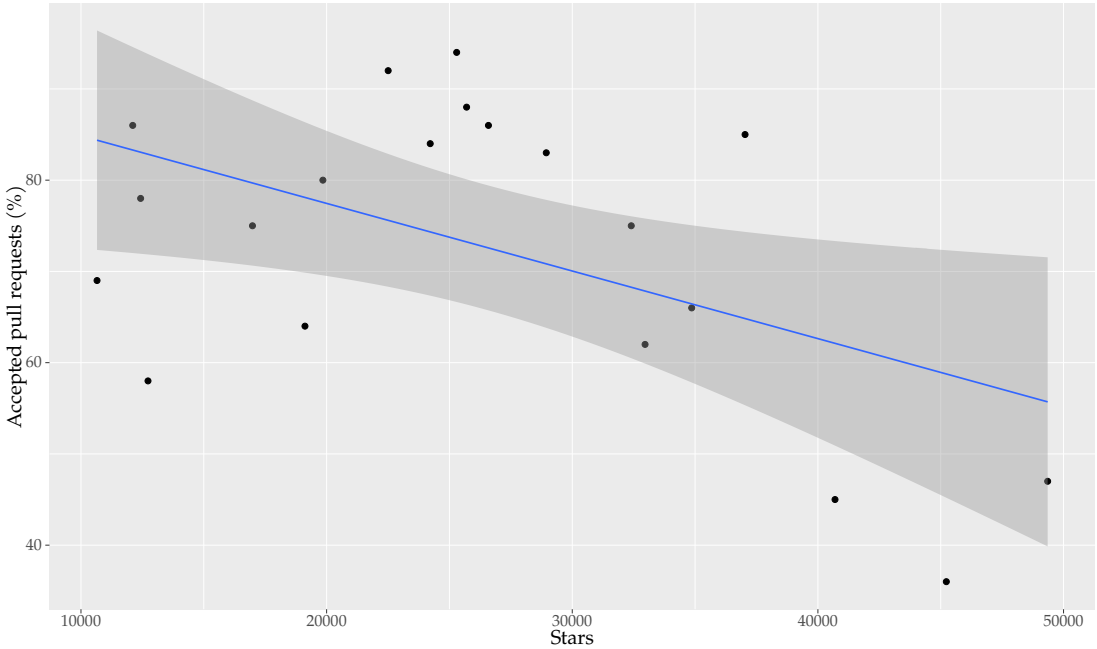

Figure 6.7: Stars and pull request acceptance

Only 1366 pull requests (from the total of 8887 pull requests) did not change the quality of the source code (did not fixed nor introduced some PMD issue). The PMD linter was able to detect 253 different issues in the given pull requests. Most of the introduced issues were issues related to the code style. In total, all of the pull requests introduced over the million of code style issues.

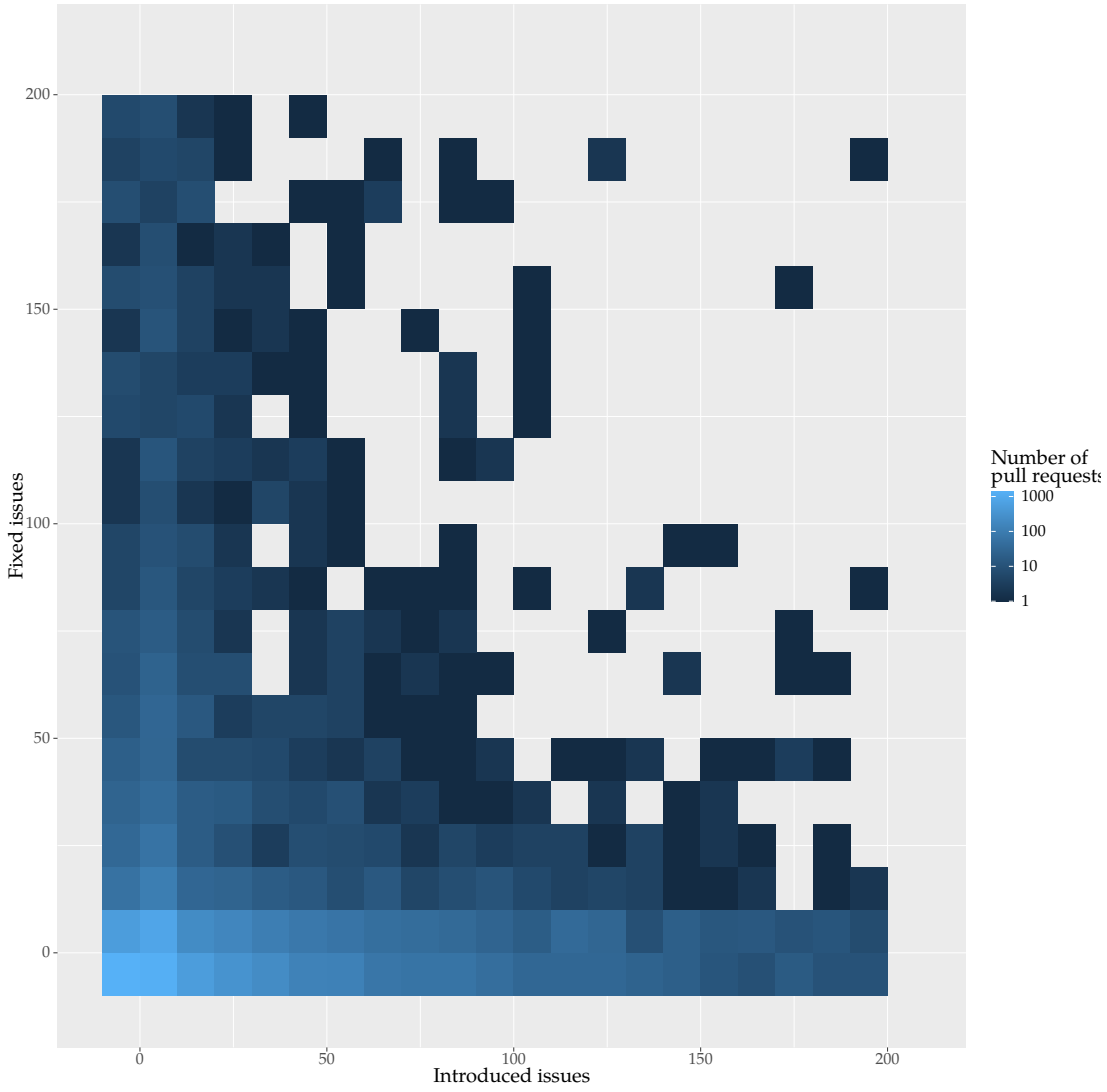

Figure 6.8: Pull requests and quality

The issue that was introduced in the largest number of pull requests is `CommentRequired` (documentation issue). Another frequent issues are `LocalVariableCouldBeFinal`, `MethodArgumentCouldBeFinal` (code style issues) and `LawOfDemeter` (issue in code design). These issues are the only issues that was introduced in more than 3000 pull requests. Similarly, the list of issues that was fixed in the largest number of pull request is dominated by the very same issues.

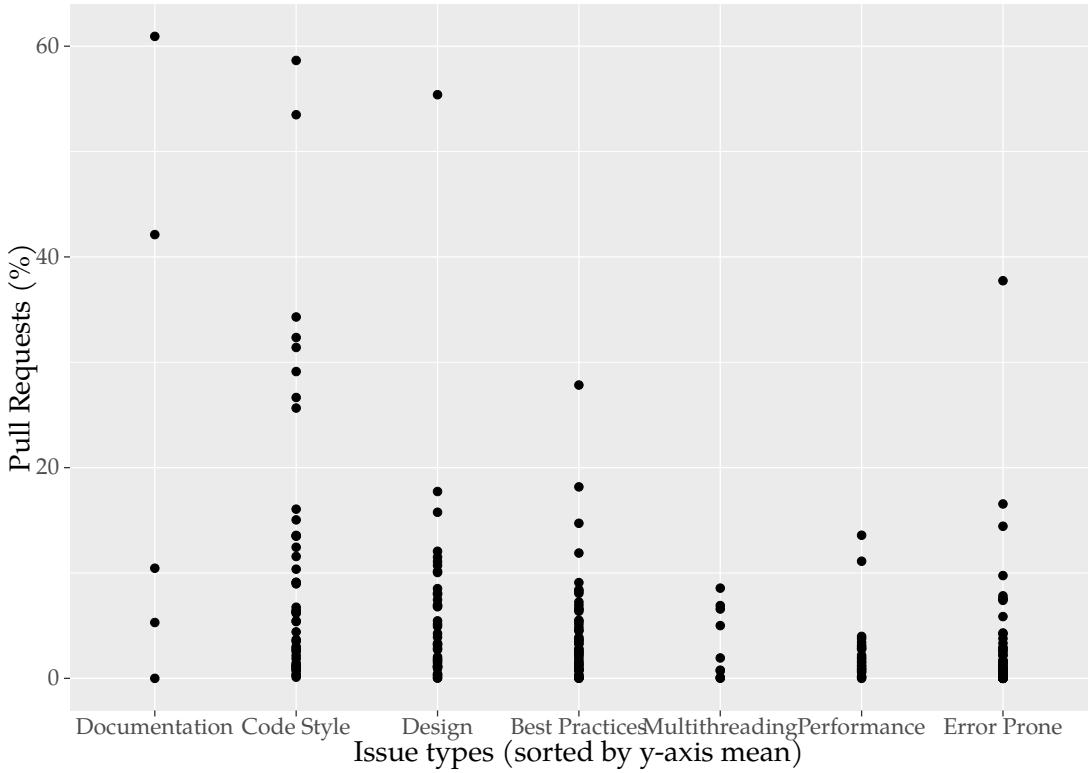

Figure 6.9: PMD issues and % of PRs which fixed/introduced them

As can be seen in Figure 6.9, the documentation issues tend to appear in large number of pull requests (24 % at average). Moreover, the typical code style issue appeared in 11 % of pull requests. On the other hand, average issue that indicate error prone construct is present in only two percent of pull requests.

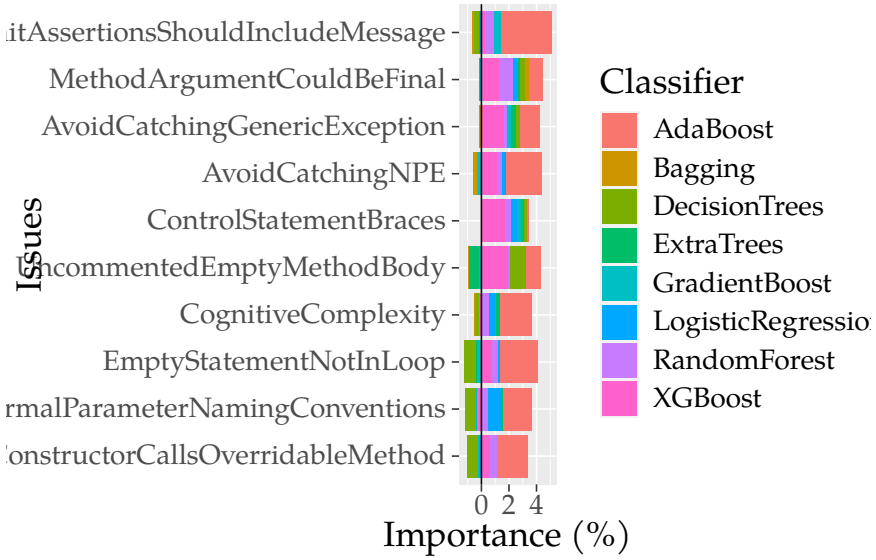

Figure 6.10: Ten most important PMD issues

The most important PMD issue is `JUnitAssertionsShouldIncludeMessage`. The average importance of this issue is only 0.6 %. However, the AdaBoost classifier gives this issue 3.7 % importance. The 0.89 issues of this type are introduced in average accepted pull request. I suspect that the pull requests that are adding larger number of tests to the codebase have higher probability to be accepted. This pull requests have also higher probability to introduce the `JUnitAssertionsShouldIncludeMessage`. This can be the reason why this issue has the largest importance. However, performed study indicates that presence of test code does not influence PR acceptance [17].

The PCA scatter plot for Java pull requests looks as follows:

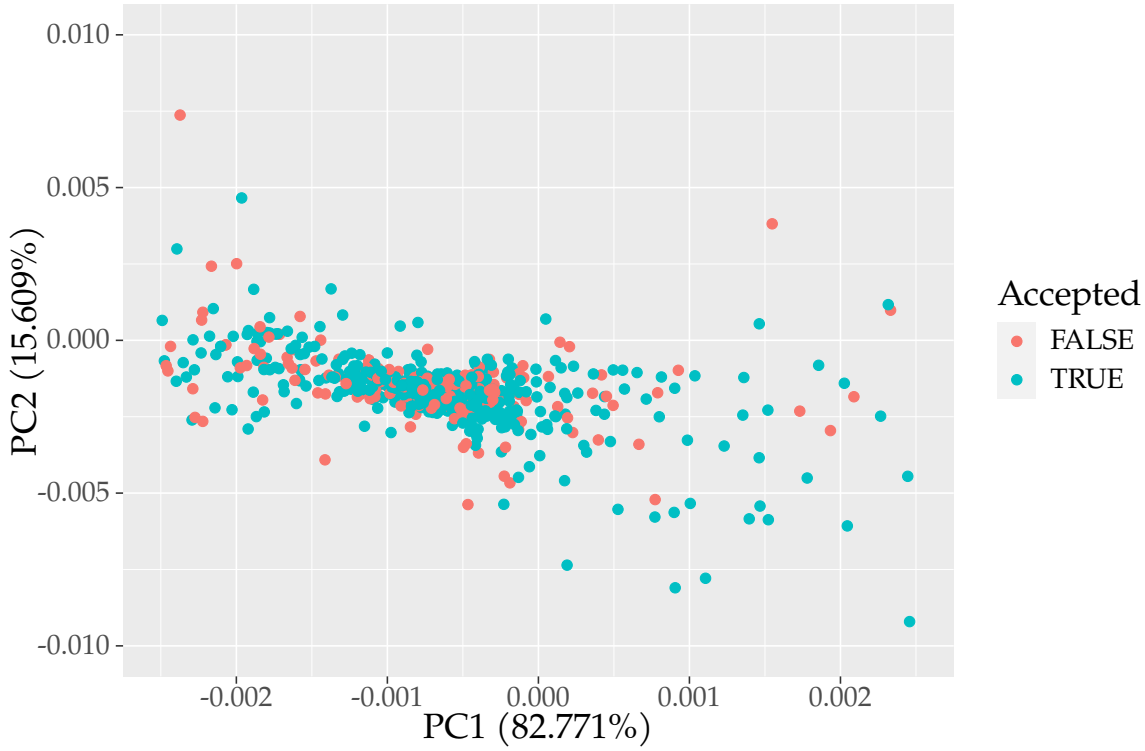

Figure 6.11: PCA scatter plot

In the PCA scatter plot, there is no visible difference between rejected/accepted pull requests.

In order to understand the relation between acceptance and the introduction of some quality issue, the chi-square test was performed.

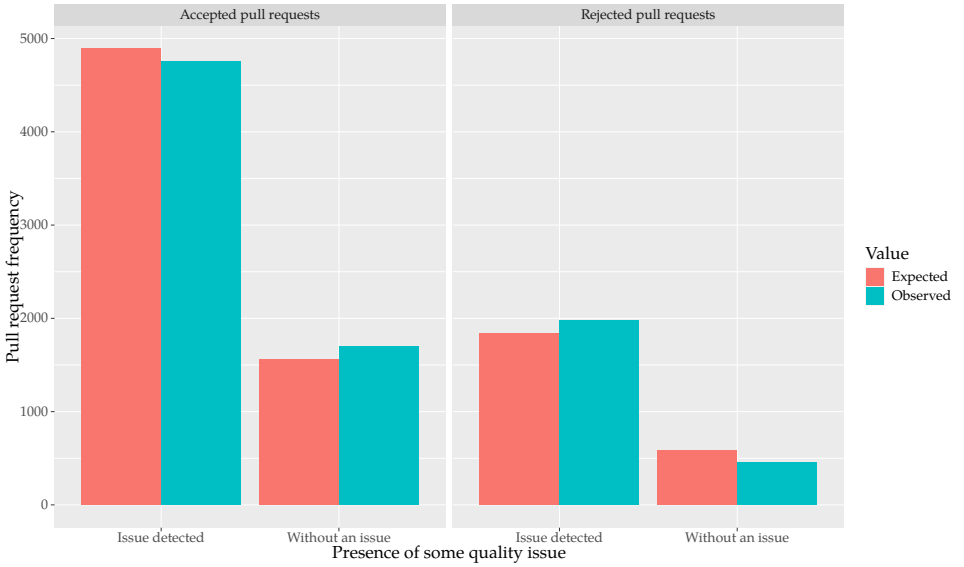

Figure 6.12: Relation between presence of issue and PR acceptance

The $p = 9.132 \times 10^{-14} < \alpha$ and $\phi_c = 0.079$; therefore, there is a weak relation between acceptance and issue presence. Similar results were obtained when only PRs that solely modified source code of the main language were considered and also when test was performed individually for each issue category.

17 out of the 20 Java projects contained sufficient number of pull requests to perform the chi-square tests. In nine of them, the code quality and acceptance are not independent. Unexpectedly, in one of the projects (`alibaba/fastjson`) the presence of some issue has small positive effect on the acceptance.

The PMD issues seems to have some effect on the time it takes to close a pull requests when considering only R^2 computed for each model. However the R^2 value is usually not a good metric how to evaluate non-linear models; it can reveal some information about the model, but it does not give us information how accurate the model is. There are three models that have $R^2 > 0.4$: Bagging, GradientBoost, RandomForest. The linear regression have $R^2 = 0.1257$, therefore for this model, 13 % of variance in time to close a PR can be explained by code issues. However, all of the models have high mean absolute error (MAE). The average MAE value for all of the models is

3934338 ≈ 46 days and 87 % of all analyzed Java pull requests was closed within one month. Therefore these models are basically useless in practice. The other models (when considering only rejected/fixed issues) yielded similar results. To conclude, the found quality issues does not have a large effect on the time to close a pull request.

6.3 Kotlin

The 20 projects were selected also from the Kotlin ecosystem. Average analyzed pull request was from project that has ten thousand stars and introduced nine issues and fixed only four. The 7514 pull requests were analyzed (using the *ktlint* linter) and 80 % of them were accepted. The trend that maintainers of popular projects reject more pull requests can be observed also in the Kotlin community.

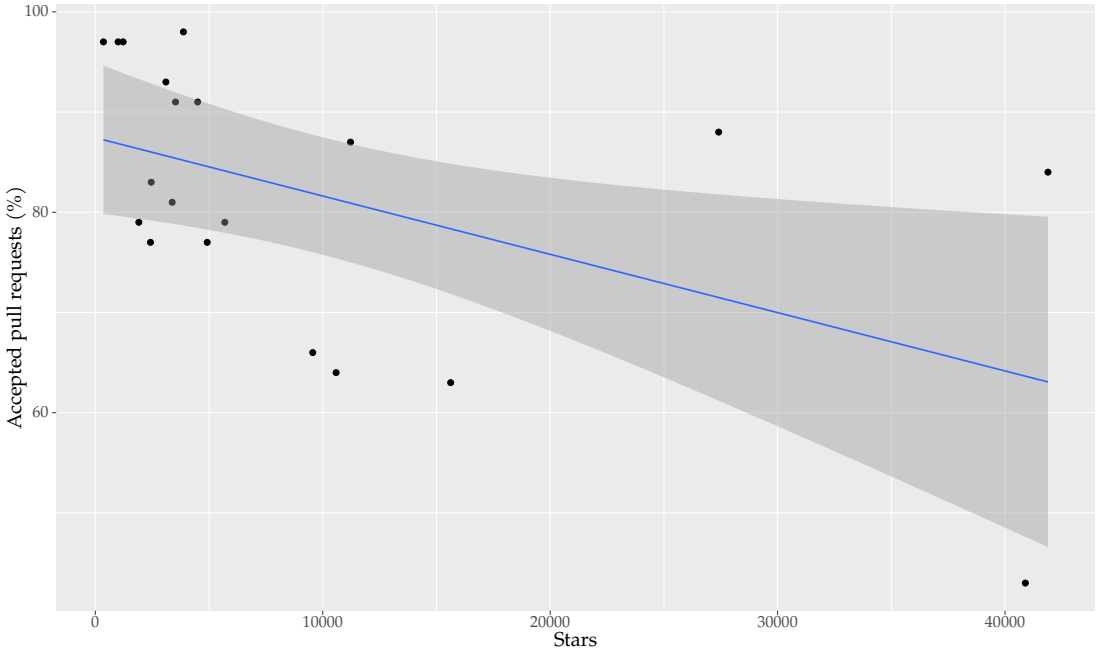

Figure 6.13: Stars and pull request acceptance

The only 20 different issues were detected by the *ktlint* in the analyzed projects; however, this is expected, since the *ktlint* is focused only on small set of quality issues.

The `indent` issue was introduced in the largest number of pull requests (2598). It is the only issue that was introduced in more than thousand pull requests. It is also the issue that was fixed in largest number of pull requests. The official Kotlin convention is to use the four spaces for indentation[1] and the `indent` issue signifies that this convention was violated. This issue influenced the code quality of more than half of the pull requests. However, this can be caused by projects whose standards does not follow the official recommendations.

Other often violated *ktlint* rules are `no-wildcard-imports`, `final-newline` and `import-ordering`. On the other end of the spectrum, the rule `no-line-break-after-else` was violated only once.

1. `https://kotlinlang.org/docs/coding-conventions.html`

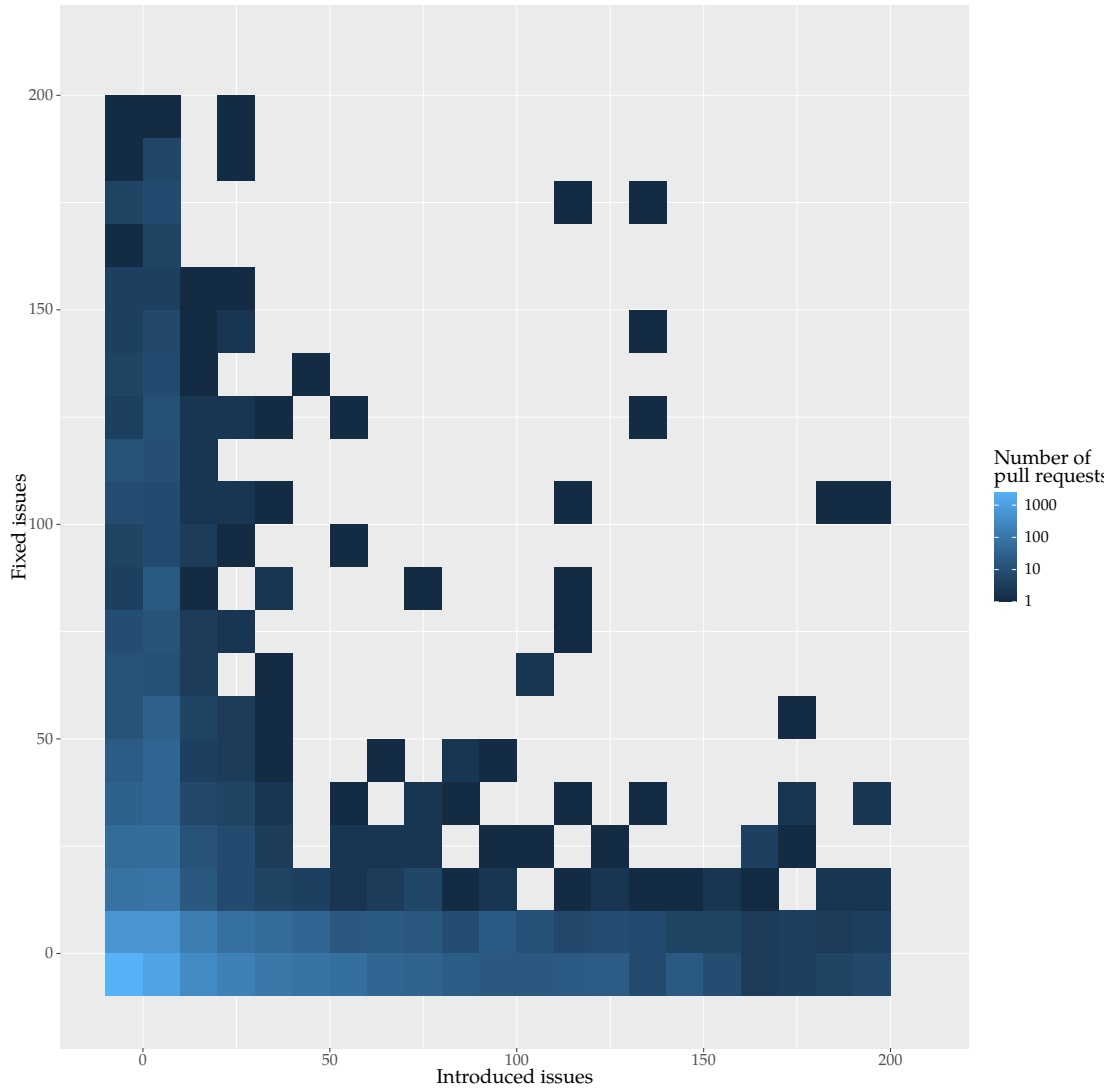

Figure 6.14: Pull requests and quality

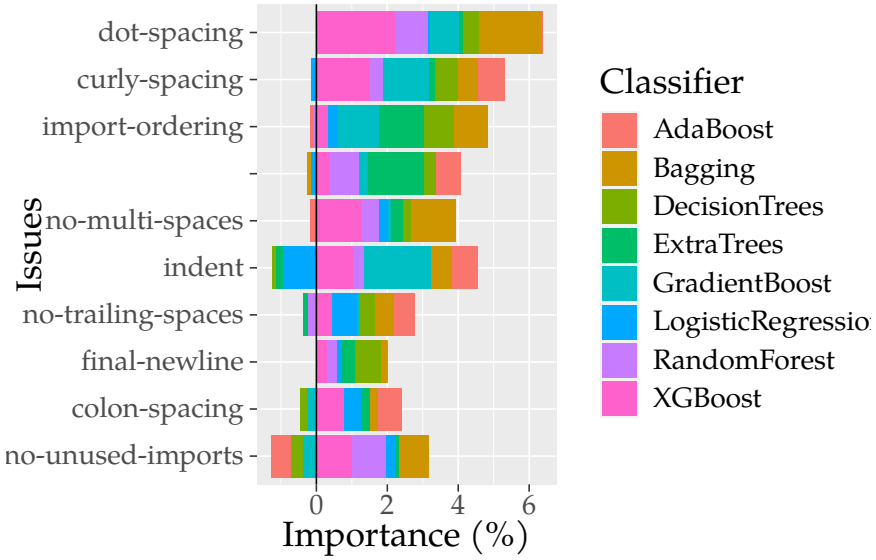

Figure 6.15: Ten most important ktlint issues

The issue with the highest importance average is `dot-spacing`. The Bagging classifier gives 1.7 % importance to this issue. The importance from other classifiers is smaller — average importance is 0.8 %. However, this issue was introduced only in 18 PRs (13 times in the rejected pull request). Furthermore, 7 accepted and 7 rejected pull requests fixed this issue.

It is worth mentioning that issue on the fourth place does not have a name (given by *ktlint*). This issue usually indicates invalid Kotlin file. This issue has high importance (relative to the other issues) also when the only fixed and also when only introduced issues were taken into account during the classification. This issue was introduced by 90 rejected PRs and by 51 accepted PRs.

When using only introduced issues, the most important issue is `indent`. This issue is also most important when only the fixed issues are considered. As being said before, in projects that are using non-standard indentation, this issue is false positive.

The PCA scatter plot was created also for Kotlin programming language. The firs principal component explains almost all variance in the code quality of pull requests. However, the difference between rejected/accepted pull requests is not apparent from the PCA plot:

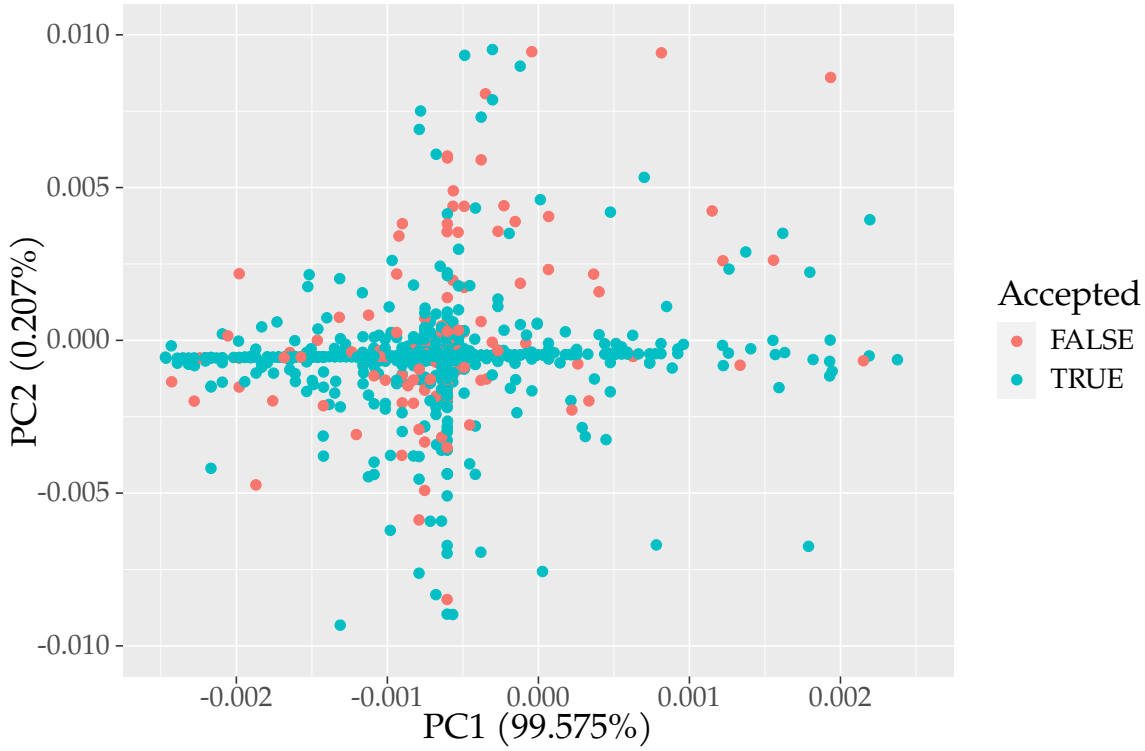

Figure 6.16: PCA scatter plot

To understand the link between acceptance and the introduction of some quality issue, the chi-square test was performed. The $p < 2.2 \times 10^{-16}$ and $\phi_c \approx 0.095$; therefore the presence of some issue has a small negative effect on acceptance (similarly to the Java and Python). Furthermore, three classifiers (*Bagging*, *GradientBoost* and *RandomForest*) have AUC for ROC curve above 60 and the average AUC is 57.58.

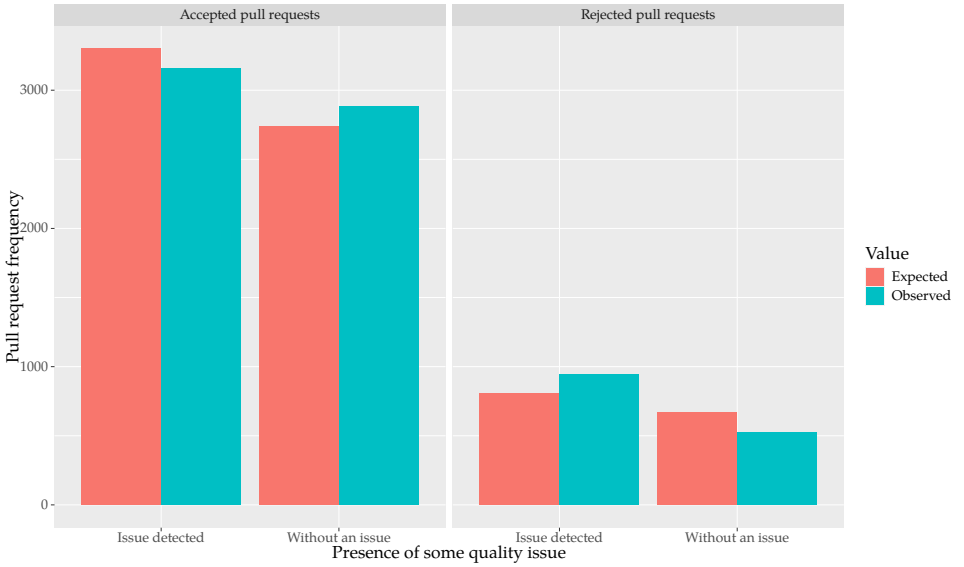

Figure 6.17: Relation between presence of issue and PR acceptance

However, taking into account solely the PRs that only modified some source code, the $p = 0.627$, thus the acceptance and issue presence are independent (in this context).

Only 12 of the projects have sufficient number of pull requests to perform the chi-square test. There are 4 projects were the presence of some issue have small impact on the PR acceptance (the average Cramer's V is $\phi_c = 0.18$).

To analyze the relation between the code quality and time that is required to close a PR, several regression techniques were applied also on the Kotlin dataset. For linear regression, $R^2 = 0.164$, therefore trained model is able to explain 16 % of the variance in the time to close a PR. The $MAE = 2375121 \approx 27$ days; therefore the model does not perform so well on the dataset, taking into consideration that 89 % of pull request were closed within one month. The mean absolute error for other models was similar to the MAE obtained for linear regression.

6.4 Haskell

Haskell is the only purely functional programming language that I analyzed. The 18 out of 20 selected Haskell projects has under the 5000 stars. There are only two exceptions: PureScript with 7632 stars and Pandoc that has over 15000 stars. The Pandoc has the also smallest percentage of accepted pull requests. However, excluding the Pandoc, there is no visible connection between number of stars and acceptance in the selected projects. When the outliers are filtered, the trend tends to be opposite of previous languages: more accepted are pull requests of projects with more stars. However, only 20 projects are not sufficient to make such conclusions about whole population of Haskell projects.

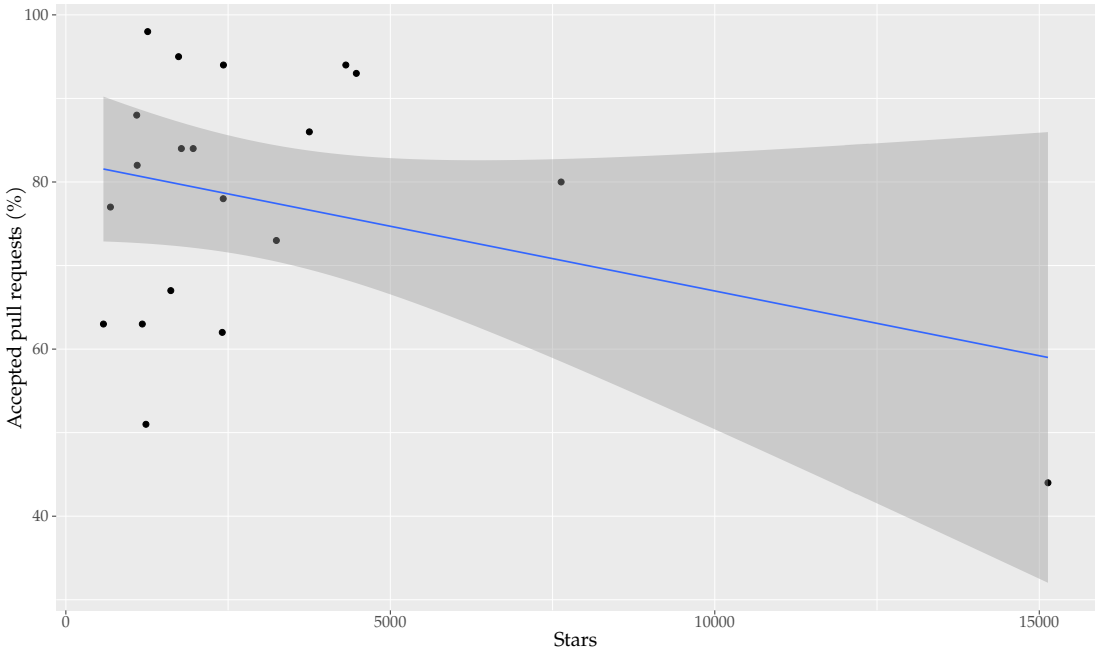

Figure 6.18: Stars and pull request acceptance

The 6949 pull request were analyzed. Interesting is that in the over than 60 % of pull requests, no change in the code quality was detected. Moreover, the *hlint* is able to recognize large number of different issues (321 issue types were detected in selected pull requests). On the other hand, some issues were counted twice because they appeared

as suggestion but also as a warning (in the different contexts). These facts can indicate that large number of submitted pull requests follows the high quality standards.

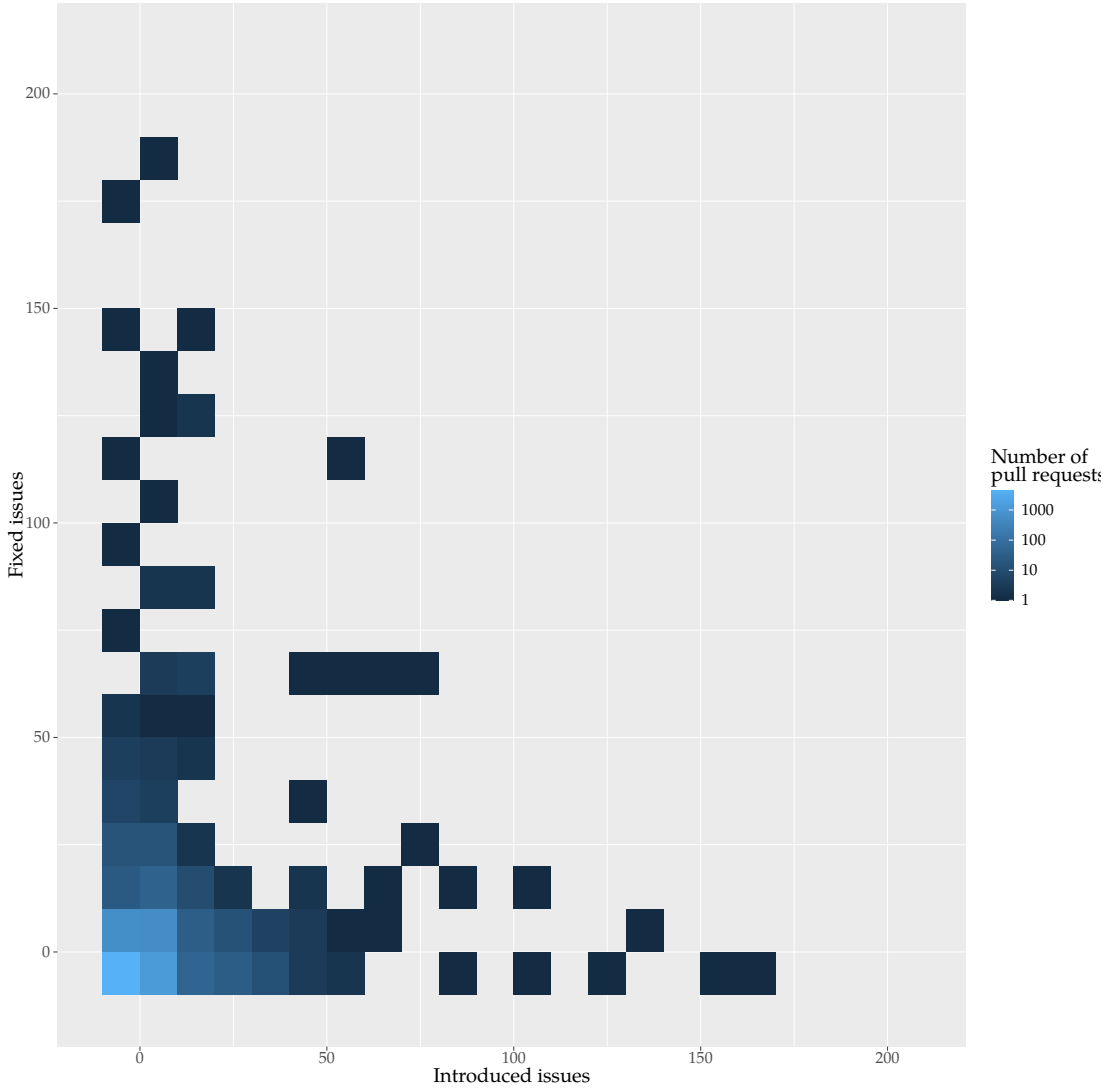

Figure 6.19: Pull requests and quality

The 78 percent of pull requests was accepted and the average pull request introduced only 0.6 issues and fixed 0.3 issues. The most common types of issues were suggestions and warnings. The error

that was introduced in the largest number of pull requests is `Use-newTVarIO` and this error was introduced only in 8 pull requests. The most common suggestions were `Redundant-bracket` (introduced in 499 PRs) and `Redundant-$` (444 PRs). The warning `Unused-LANGUAGE-pragma` was introduced in 323 pull requests and `Eta-reduce` warning in 214 of them. There were only ten issues that were introduced in 100 and more pull requests. 105 issue types were detected in the analyzed code but no PR introduce any of those issues.

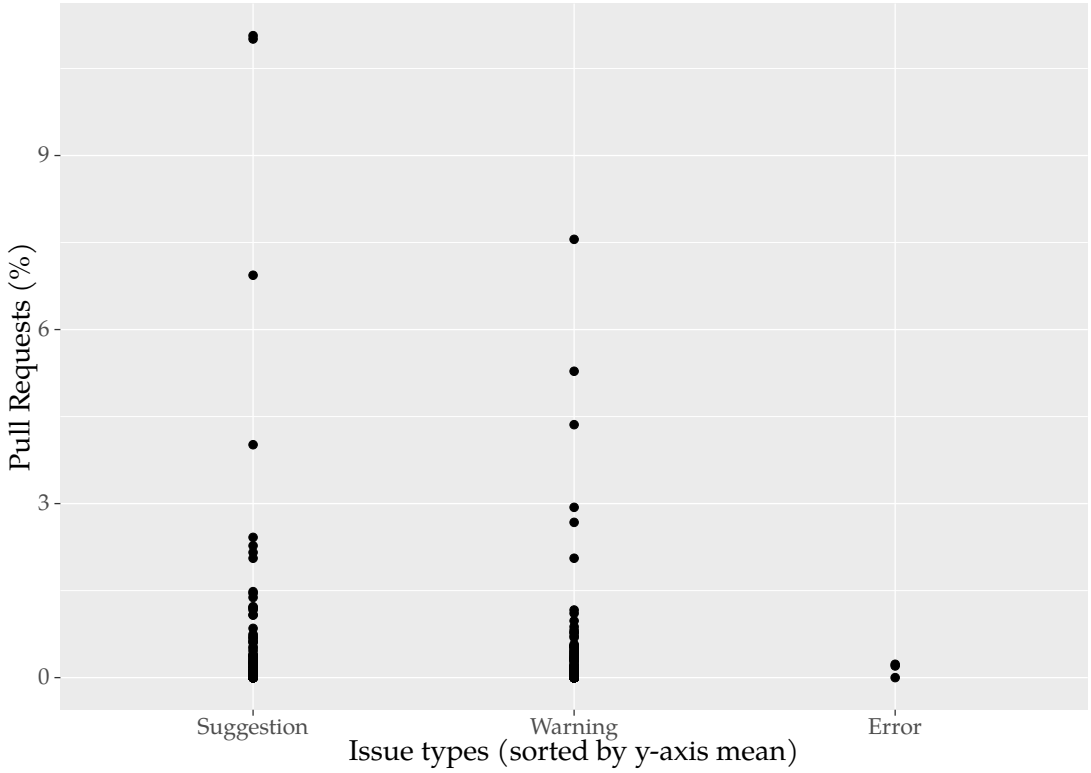

Figure 6.20: HLint issues and % of PRs which fixed/introduced them

The most important Haskell issue is suggestion `Use-if`. However, no classifier gives this issue importance over one percent. Therefore the actual impact of this issue is disputable. This issue was introduced in 18 rejected PRs and fixed in 11. There are 19 accepted PRs that introduced `Use-if` and 27 accepted PRs that fixed it.

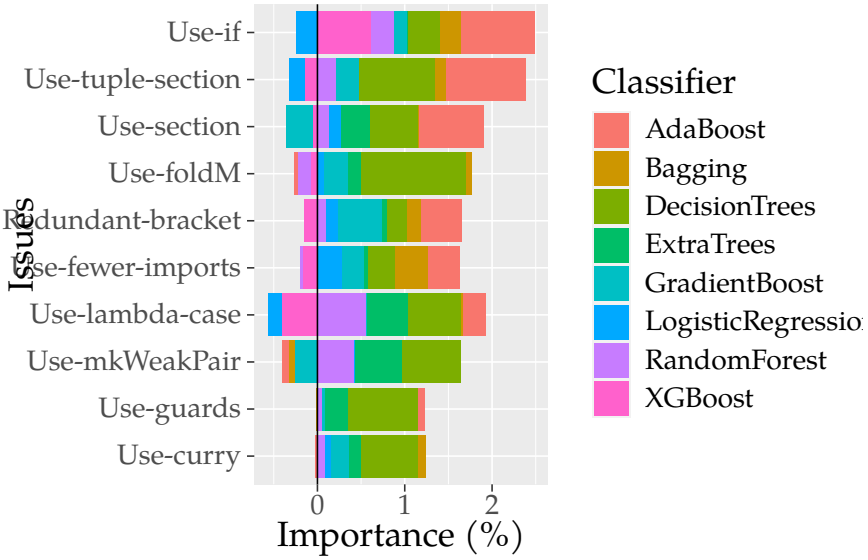

Figure 6.21: Ten most important HLint issues

When only introduced issues were taken into account, the most important issues is `Move-brackets-to-avoid-$` (suggestion). The AdaBoost classifier gives this issue 1 % importance, although the average importance is only 0.4 %.

In context of fixed issues, the most important is warning `Use-fewer-imports` with average importance again only about 0.4 %.

The PCA scatter plot was also generated for the Haskell language. Similarly to the results in already analyzed languages, there is no apparent difference between accepted and rejected pull requests.

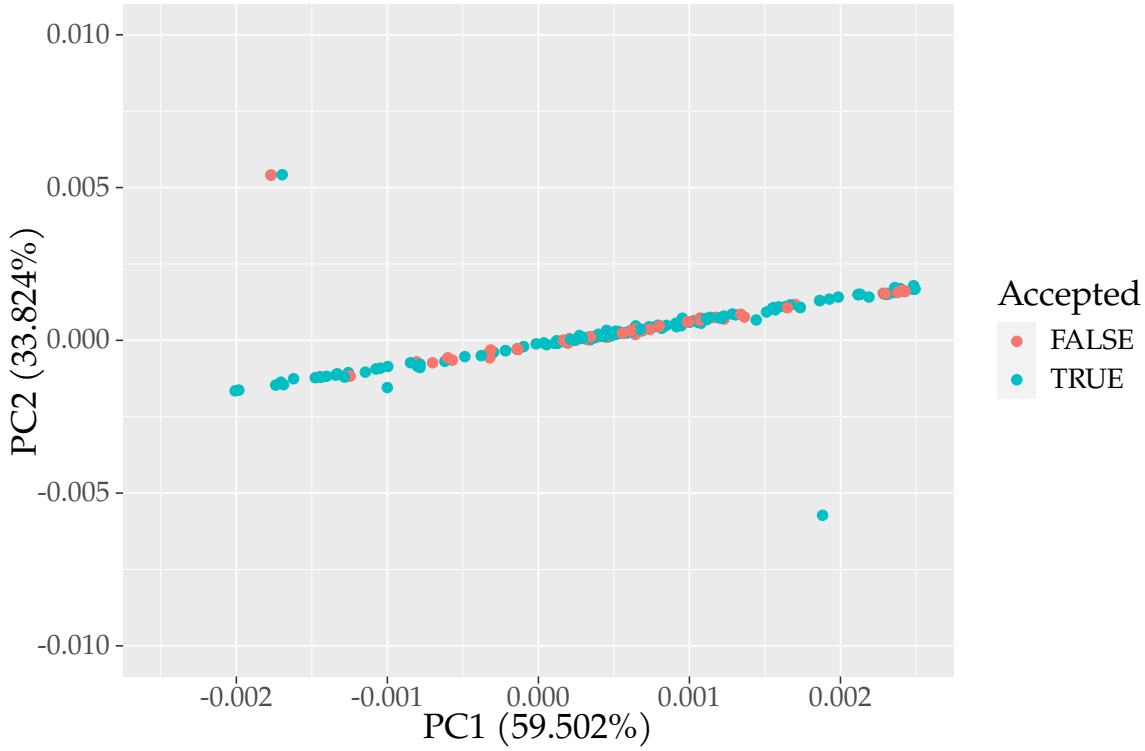

Figure 6.22: PCA scatter plot

For the chi-square test, the $p = 0.001438 < \alpha = 0.05$ and Cramer's V is only $\phi_c = 0.038$; therefore, the presence of an issue in the PR's have only small negative effect on the acceptance of the pull request. Similar results were obtain when only the pull requests, that contains exclusively some modified code, were considered. Furthermore, tests for the individual issue types yielded also similar results. Unfortunately, there is only small number of pull requests that introduced some errors; therefore the chi-square test cannot be performed on this issue category. The average AUC computed for ROC curves is around 50 — the classification algorithms were unable to find some significant association between the acceptance and code quality issues.

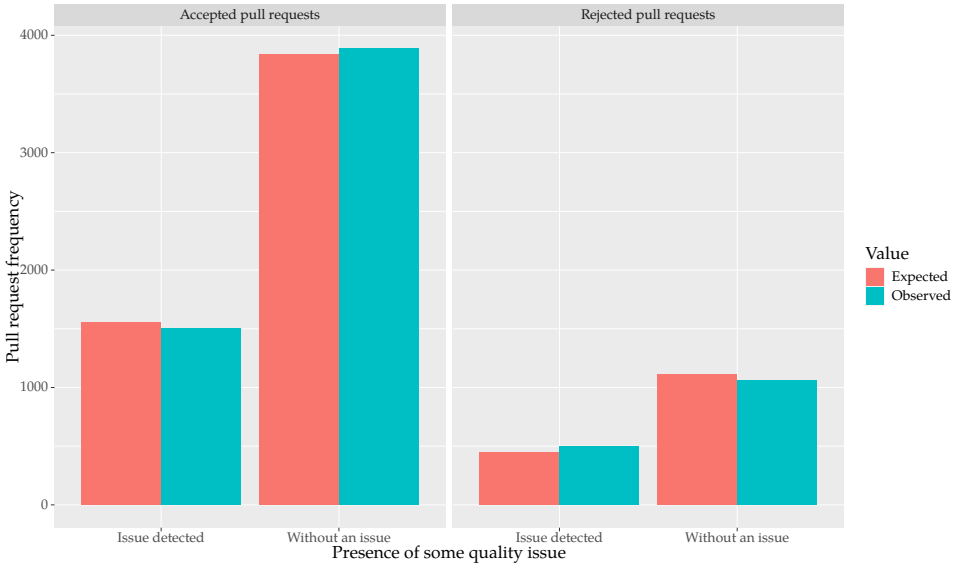

Figure 6.23: Relation between presence of issue and PR acceptance

The 13 projects contains sufficient number of pull requests; the acceptance and the issue presence is not independent only in four of them (the issue presence have the small negative impact on the acceptance). For the `haskell/aeson` project, the Cramer's V is 0.282 — the association is "medium".

The issues detected by *HLint* does not seems to have impact on time it takes to close a pull request. All trained models have negative R^2. When only fixed issues were used for regression, there were three models with positive R^2: Bagging (0.0315), ElasticNet (0.0085) and RandomForest (0.0229). However, all of them have high mean absolute error: Bagging (2193658 $\approx$ 25 days), ElasticNet (2255678) and RandomForest (2201347).

6.5 C/C++

The C and C++ programming languages are analyzed together because they share lot of similarities. This usually enables to use the same linter for both languages. Moreover, it is not uncommon that projects that are written in C++ contains also some C code and vice

versa. The 9 selected projects has more code written in C, while the rest of the 11 projects is more C++-oriented.

In analyzed projects, there is no visible connection between the acceptance and the number of stars.

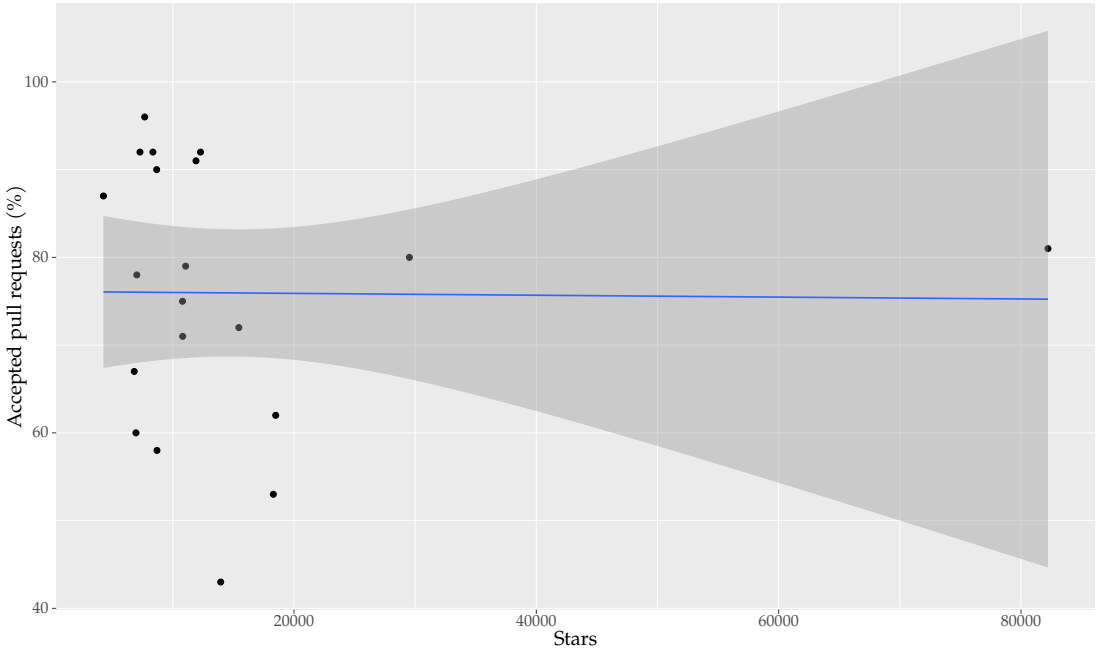

Figure 6.24: Stars and pull request acceptance

I analyzed 8774 C/C++ pull requests. The 77 % of them are accepted pull requests. The typical pull request introduces 0.25 issues and fixes 0.12 issues, the typical rejected PR introduces 0.79 issues and typical accepted PR only 0.15 issues. The 79 % of pull request did not change the quality of the source code (in terms of the *flawfinder* quality rules).

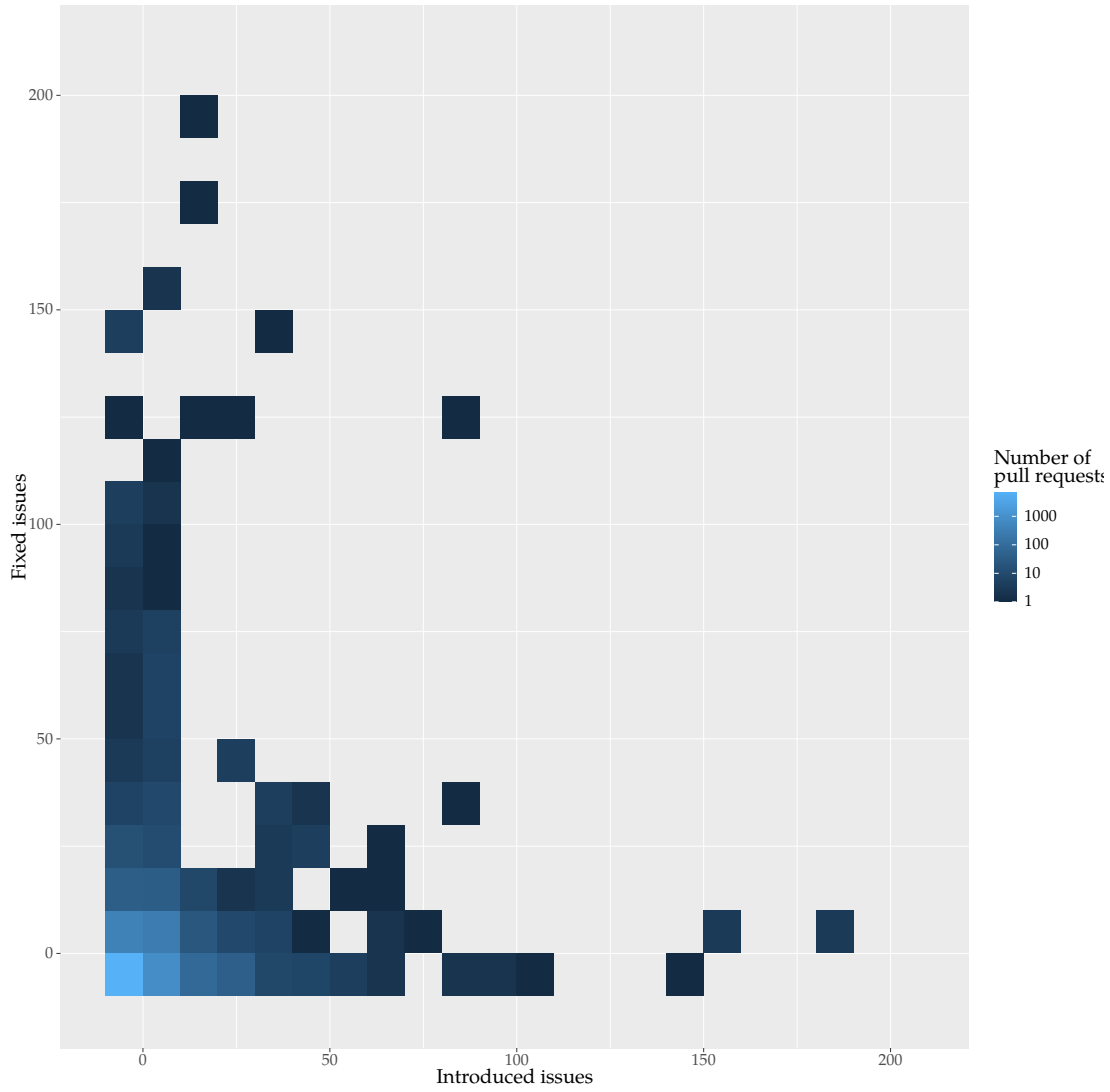

Figure 6.25: Pull requests and quality

The most common type of issue is note. Least common are errors. The *flawfinder* was able to identify 137 different issues in the studied PRs. All of the top ten issues (in terms of umber of PRs which introduced them) are notes. The most common note is `buffer-char` ("Statically-sized arrays can be improperly restricted leading to potential overflows or other issues…"). The most common error is `buffer-`

`strcat` ("Does not check for buffer overflows when concatenating to destination…") and it is the 11 most introduced issue (introduced in 69 pull requests). There are 36 issues that were present in the analyzed code but they were not introduced in any pull request; 21 of them are errors.

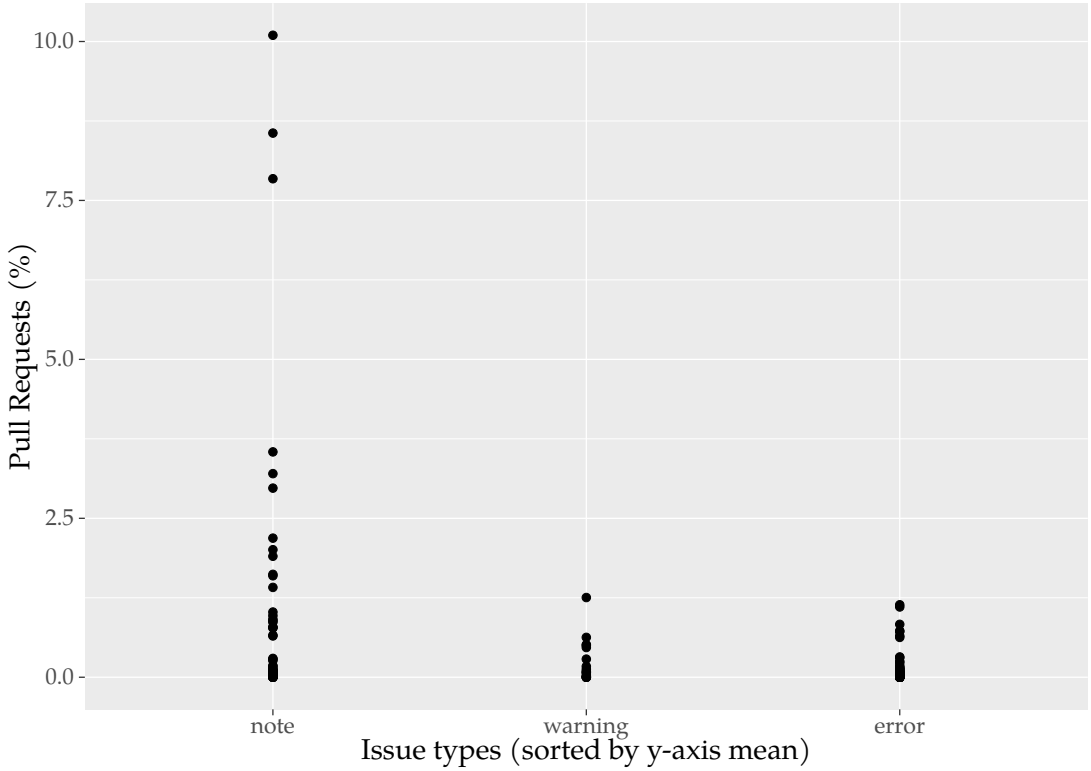

Figure 6.26: flawfinder issues and % of PRs which fixed/introduced them

Classification algorithms ranks as the most important issue the `format-printf` ("If format strings can be influenced by an attacker, they can be exploited…"). However, this issue is only a *note*. Therefore it does not have to indicate a defect (there will probably be large number of false positives). AdaBoost and XGBoost algorithms gives this issue importance of 1 %. The average importance is 0.7%. This issue is also most important when only introduced issues are considered. Second most important issue has average importance only 0.26 %.

The most important error is `buffer-StrCpyNA` ("Does not check for buffer overflows when copying to destination...") with average importance only 0.9 %. This error is sixth most important issue.

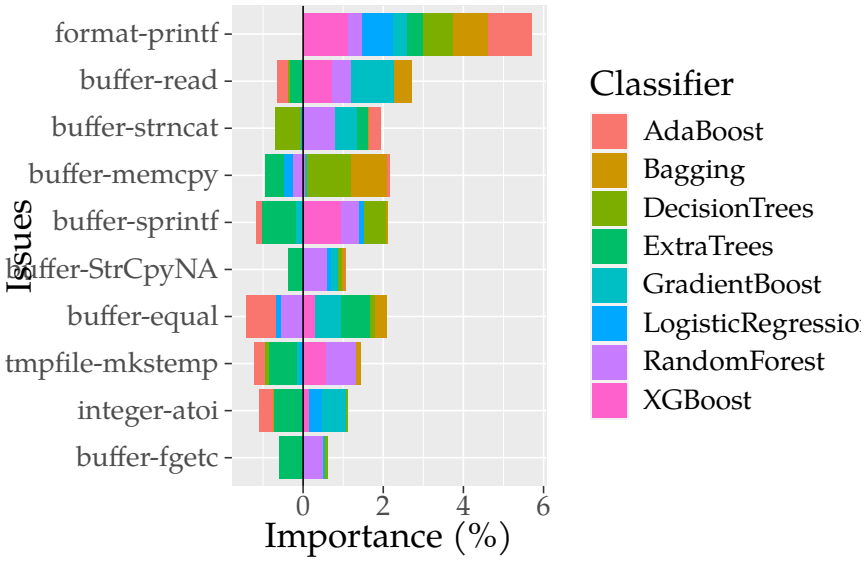

Figure 6.27: Ten most important flawfinder issues

When considering only fixed issues, the `buffer-read` is the most important issue (note); however, the average importance is only 0.28 %.

The PCA analysis does not reveal any significant difference between the accepted and rejected pull requests (in terms of code quality).

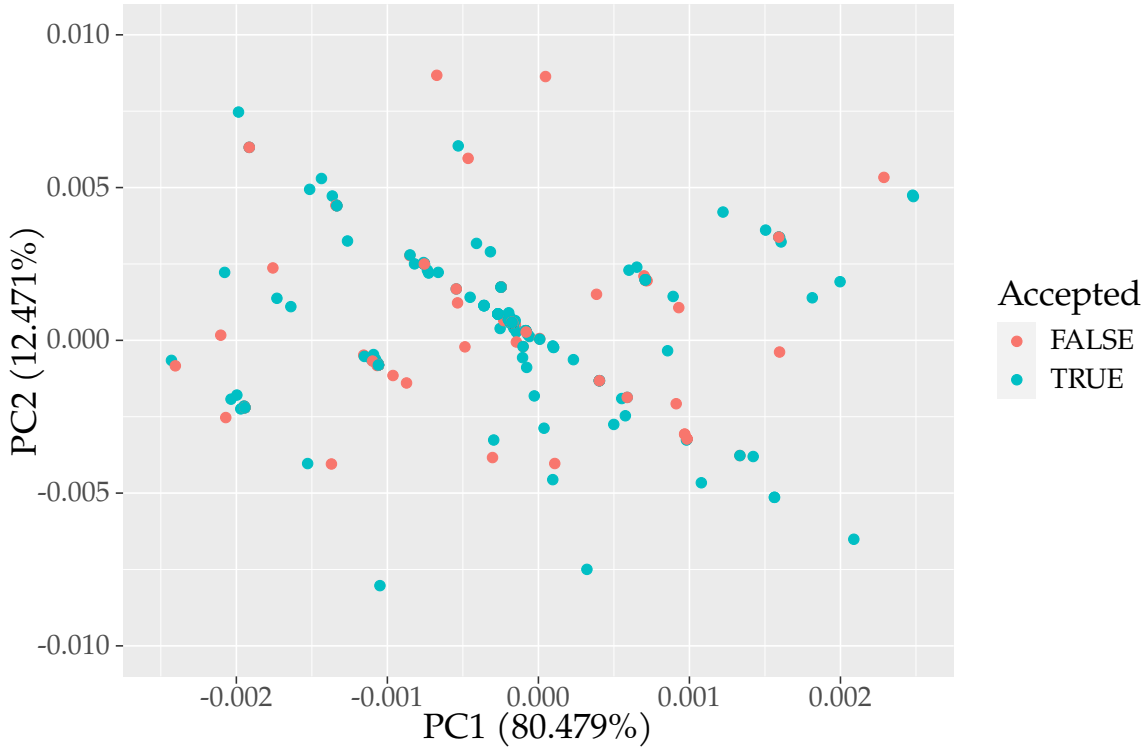

Figure 6.28: PCA scatter plot

Based on the chi-square test, the presence of issue in the PR have small negative impact on the PR acceptance ($\phi_c = 0.117$). When considering only pull requests that solely modified some source files, Cramer's V $\phi_c = 0.024$ — the presence of issue also have some negative impact but even smaller than in the previous test.

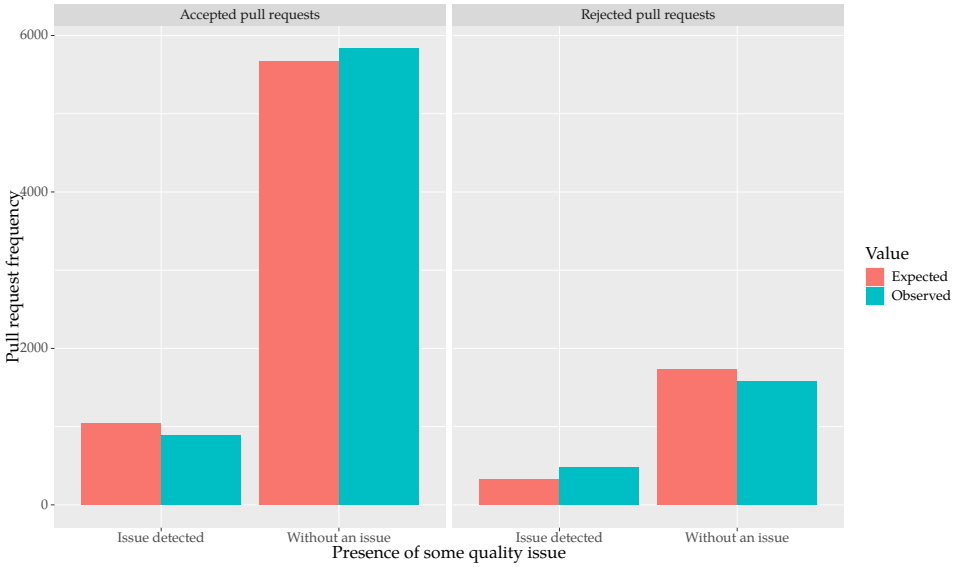

Figure 6.29: Relation between presence of issue and PR acceptance

Some small impact was also discovered when the chi-square test was performed separately for each issue category (the $p < 2.2 \times 10^{-16}$ and $\phi_c \approx 0.1$ for each category). Furthermore, in 6 out of 11 projects, which have enough data to perform and evaluate the chi-square test, the presence of some issue in the PR have negative effect on the PR acceptance. In the `minetest/minetest` and `pybind/pybind11` projects this effect is moderate; for other projects the association is small.

In case of C/C++, the time to close a pull request seems not to be related with found issues. All the models have negative R^2, except the ElasticNet regressor. For the ElasticNet, $MAE = 4681624$ (mean absolute error is 54 days) — therefore, this model also cannot be used to predict the time to close a PR. Models considering only rejected issues and also models considering only accepted issues have yielded similar results.

6.6 Programming languages and code quality impact

Comparing the code quality of projects written in different programming languages is difficult task. Each language have different programming constructs, syntax and type system. For instance, Python,

which is dynamically-typed multi-paradigm programming language, has completely distinct characteristics than Haskell, which is purely functional programming language with strong, static type system.

Moreover, every linter is different and have unique set of rules. The *ktlint* is focused on code clarity and community conventions whereas *flawfinder* checks code for potentially dangerous functions. On the other hand, the *PMD* is more general-oriented linter that contains large set of rules for Java programming language. Lastly, the *HLint* is oriented mainly on code simplification and spotting of redundancies.

On the other hand, there are some metrics that evaluates how effectively can trained models predict the acceptance of PR/time to close a PR; and these metrics can be compared across different programming language. On top of that, the results from chi-square test can be also compared. However, cation is in order because the quality for each language is evaluated differently as discussed before.

6.7 TODO Threads to validity

- PR recognition (rejected PRs can be merged using another way)
  - manually inspect some randomly selected PRs
- modified files in the PR's that are not written in the primary programming language can influence acceptance
- chosen projects
- PR's filtering
  - linter errors
  - limit of 500 PRs
  - limited execution time for `git-contrast`
  - not available PR's/repositories
- false positives from Linters (`import-error`, `relative-beyond-top-level`)

7 TODO Conclusion

7.1 TODO Comparison with the previous results

7.2 TODO Future work

Appendix

Projects

Table 7.1: Python projects

Project	Stars	Analyzed PRs	Accepted	Rejected	Introduced issues	Fixed issues
pallets/flask	58380	500	76%	24%	2.82	1.88
rg3/youtube-dl	50768	808	49%	51%	5.34	2.11
psf/requests	47100	500	64%	36%	2.76	1.04
nvbn/thefuck	46148	268	47%	53%	6.61	0.98
scrapy/scrapy	43124	500	80%	20%	5.63	5.39
faif/python-patterns	31006	258	90%	10%	4.20	5.30
certbot/certbot	28785	500	81%	19%	3.62	1.60
openai/gym	26986	500	68%	32%	7.12	3.05
soimort/you-get	25437	487	56%	44%	7.77	1.91
explosion/spaCy	23007	500	91%	9%	3.68	3.79
pypa/pipenv	22785	500	86%	14%	5.62	1.78
keon/algorithms	20528	341	81%	19%	11.42	9.82
tornadoweb/tornado	20451	500	80%	20%	3.18	1.36
keras-team/keras	20384	398	53%	47%	4.88	3.49
celery/celery	18850	500	76%	24%	3.60	1.24
locustio/locust	18518	496	74%	26%	9.02	3.90
sanic-org/sanic	15958	500	81%	19%	4.86	1.95
spotify/luigi	15485	500	72%	28%	5.62	2.96
kivy/kivy	14471	500	89%	11%	2.50	1.18
powerline/powerline	13187	396	81%	19%	23.88	4.10

Issue categories

Model reliability

- for regression and classification
- check the Lenarduzzi paper
- table with recall, precision etc.

Table 7.2: Java projects

Project	Stars	Analyzed PRs	Accepted	Rejected	Introduced issues	Fixed issues
iluwatar/java-design-patterns	49353	258	47%	53%	47.50	56.09
TheAlgorithms/Java	45228	430	36%	64%	22.60	96.09
ReactiveX/RxJava	40697	722	45%	55%	64.45	9.55
apache/dubbo	37035	500	85%	15%	23.98	44.06
PhilJay/MPAndroidChart	34862	251	66%	34%	18.04	8.97
square/retrofit	32964	493	62%	38%	15.80	4.66
bumptech/glide	32402	343	75%	25%	15.06	2.64
netty/netty	28942	500	83%	17%	15.75	12.72
apolloconfig/apollo	26588	500	86%	14%	29.91	2.23
JakeWharton/butterknife	25699	243	88%	12%	22.94	5.96
alibaba/druid	25294	500	94%	6%	22.05	20.24
alibaba/fastjson	24218	443	84%	16%	19.45	9.24
Netflix/Hystrix	22506	500	92%	8%	37.28	10.98
libgdx/libgdx	19848	500	80%	20%	10.99	5.84
google/ExoPlayer	19119	500	64%	36%	62.70	49.39
mybatis/mybatis-3	16982	500	75%	25%	22.13	7.71
arduino/Arduino	12729	500	58%	42%	1702.66	100.61
apache/hadoop	12429	500	78%	22%	20.44	25.30
pinpoint-apm/pinpoint	12107	500	86%	14%	54.86	31.20
android-async-http/android-async-http	10654	204	69%	31%	10.41	5.33

Table 7.3: Kotlin projects

Project	Stars	Analyzed PRs	Accepted	Rejected	Introduced issues	Fixed issues
square/okhttp	41886	430	84%	16%	31.24	26.01
JetBrains/kotlin	40892	500	43%	57%	5.83	10.22
square/leakcanary	27408	300	88%	12%	40.75	6.66
tachiyomiorg/tachiyomi	15623	252	63%	37%	11.07	1.41
android/compose-samples	11220	300	87%	13%	0.06	0.00
Kotlin/kotlinx.coroutines	10586	398	64%	36%	11.38	10.02
ktorio/ktor	9559	480	66%	34%	4.64	2.12
mozilla-mobile/fenix	5694	500	79%	21%	0.15	0.05
arrow-kt/arrow	4918	442	77%	23%	29.69	2.00
cashapp/sqldelight	4497	392	91%	9%	12.70	3.46
intellij-rust/intellij-rust	3873	500	98%	2%	5.94	20.64
gradle/kotlin-dsl-samples	3526	278	91%	9%	8.02	2.62
kotest/kotest	3378	341	81%	19%	33.86	9.02
square/kotlinpoet	3103	401	93%	7%	16.96	1.68
edvin/tornadofx	2457	254	83%	17%	5.57	1.15
Kotlin/dokka	2424	446	77%	23%	7.90	10.62
mozilla-mobile/android-components	1913	500	79%	21%	0.46	0.11
DroidKaigi/conference-app-2018	1222	292	97%	3%	3.98	8.42
JetBrains/kotlin-wrappers	1000	294	97%	3%	0.84	0.20
wordpress-mobile/AztecEditor-Android	356	214	97%	3%	7.13	1.95

Table 7.4: Haskell projects

Project	Stars	Analyzed PRs	Accepted	Rejected	Introduced issues	Fixed issues
jgm/pandoc	15134	361	44%	56%	1.20	0.59
purescript/purescript	7632	456	80%	20%	0.48	0.15
carp-lang/Carp	4476	340	93%	7%	0.62	0.17
unisonweb/unison	4314	436	94%	6%	0.64	0.49
input-output-hk/cardano-sl	3752	488	86%	14%	1.99	2.43
commercialhaskell/stack	3244	519	73%	27%	0.33	0.20
haskell/haskell-ide-engine	2428	405	94%	6%	0.55	0.39
wireapp/wire-server	2425	223	78%	22%	1.51	0.53
yesodweb/yesod	2410	336	62%	38%	0.23	0.04
simonmichael/hledger	1962	262	84%	16%	0.62	0.44
agda/agda	1780	298	84%	16%	0.80	0.56
diku-dk/futhark	1737	287	95%	5%	0.24	0.10
ekmett/lens	1617	200	67%	33%	0.11	0.00
ndmitchell/hlint	1261	301	98%	2%	0.04	0.00
haskell-servant/servant	1234	217	51%	49%	0.43	0.09
haskell/cabal	1178	847	63%	37%	0.22	0.14
haskell/aeson	1099	246	82%	18%	0.30	0.06
clash-lang/clash-compiler	1092	241	88%	12%	1.00	0.36
ucsd-progsys/liquidhaskell	687	253	77%	23%	2.37	0.85
yesodweb/wai	579	233	63%	37%	0.27	0.11

Table 7.5: C/C++ projects

Project	Stars	Analyzed PRs	Accepted	Rejected	Introduced issues	Fixed issues
microsoft/terminal	82226	500	81%	19%	0.00	0.02
nlohmann/json	29526	353	80%	20%	0.02	0.00
nothings/stb	18491	368	62%	38%	0.08	0.00
mpv-player/mpv	18296	500	53%	47%	0.25	0.38
simdjson/simdjson	15446	413	72%	28%	0.67	0.24
micropython/micropython	13951	500	43%	57%	0.14	0.08
hashcat/hashcat	12288	500	92%	8%	0.72	0.14
Tencent/rapidjson	11910	392	91%	9%	0.10	0.00
davisking/dlib	11058	275	79%	21%	0.04	0.03
reactos/reactos	10812	500	71%	29%	0.15	0.12
pybind/pybind11	10799	500	75%	25%	0.06	0.04
libevent/libevent	8693	293	58%	42%	0.34	0.79
irungentoo/toxcore	8667	500	90%	10%	0.85	0.75
libgit2/libgit2	8356	500	92%	8%	0.33	0.22
zeromq/libzmq	7678	500	96%	4%	0.52	0.33
Z3Prover/z3	7287	500	92%	8%	0.01	0.00
nodemcu/nodemcu-firmware	7028	500	78%	22%	1.48	0.93
minetest/minetest	6956	500	60%	40%	1.75	0.10
microsoft/cpprestsdk	6815	180	67%	33%	0.12	0.01
sass/libsass	4273	500	87%	13%	0.00	0.00

Table 7.6: Pylint issue categories

Category	Introduced in total	#PRs which introduced	Fixed in total	#PRs which fixed
warning	48931	3350	36865	1910
error	24657	2540	18841	1310
convention	91770	4683	76324	2447
refactor	16317	2543	14964	1483
info	2	1	2	1

Table 7.7: PMD issue categories

Category	Introduced in total	#PRs which introduced	Fixed in total	#PRs which fixed
Code Style	1341883	5780	522255	3387
Design	189046	4689	212123	2617
Documentation	343493	4524	222088	2032
Error Prone	157278	3435	60056	2062
Multithreading	9475	938	7818	825
Best Practices	177369	3822	128873	2133
Performance	15125	1512	18002	1092

ROC curves

Bibliography

1. YU, Y.; WANG, H.; FILKOV, V.; DEVANBU, P.; VASILESCU, B. Wait for It: Determinants of Pull Request Evaluation Latency on GitHub. In: *Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 367–371. Available from DOI: 10.1109/MSR.2015.42.
2. TRAUTSCH, A.; HERBOLD, S.; GRABOWSKI, J. A longitudinal study of static analysis warning evolution and the effects of PMD on software quality in Apache open source projects. *Empirical Software Engineering*. 2020, vol. 25, pp. 5137–5192. Available from DOI: 10.1007/s10664-020-09880-1.
3. GOUSIOS, G.; ZAIDMAN, A.; STOREY, M.; DEURSEN, A. Work Practices and Challenges in Pull-Based Development: The Integrator's Perspective. In: *International Conference on Software Engineering*. IEEE, 2015, vol. 1, pp. 358–368. Available from DOI: 10.1109/ICSE.2015.55.
4. KONONENKO, O.; ROSE, T.; BAYSAL, O.; GODFREY, M.; THEISEN, D.; WATER, B. Studying Pull Request Merges: A Case Study of Shopify's Active Merchant. In: *International Conference on Software Engineering*. ACM, 2018, pp. 124–133. Available from DOI: 10.1145/3183519.3183542.
5. LI, Z.; YU, Y.; WANG, T.; YIN, Gang; LI, Shanshan; WANG, Huaimin. Are You Still Working on This An Empirical Study on Pull Request Abandonment. *Transactions on Software Engineering*. 2021, pp. 1–1. Available from DOI: 10.1109/TSE.2021.3053403.
6. ALAMI, A.; COHN, L.; WĄSOWSKI, A. How Do FOSS Communities Decide to Accept Pull Requests? In: *Proceedings of the Evaluation and Assessment in Software Engineering*. ACM, 2020, pp. 220–229. Available from DOI: 10.1145/3383219.3383242.
7. TSAY, J.; DABBISH, L.; HERBSLEB, J. Influence of Social and Technical Factors for Evaluating Contribution in GitHub. In: *International Conference on Software Engineering*. ACM, 2014, pp. 356–366. Available from DOI: 10.1145/2568225.2568315.

8. SOARES, D.; DE LIMA, M.; MURTA, L.; PLASTINO, A. Acceptance Factors of Pull Requests in Open-Source Projects. In: *Symposium on Applied Computing*. ACM, 2015, pp. 1541–1546. Available from DOI: 10.1145/2695664.2695856.

9. DEY, T.; MOCKUS, A. Effect of Technical and Social Factors on Pull Request Quality for the NPM Ecosystem. In: *International Symposium on Empirical Software Engineering and Measurement*. ACM, 2020. Available from DOI: 10.1145/3382494.3410685.

10. JOSH, J.; KOFINK, A.; MIDDLETON, J.; RAINEAR, C.; MURPHY-HILL, E.; PARNIN, C.; STALLINGS, J. Gender differences and bias in open source: Pull request acceptance of women versus men. *PeerJ Computer Science*. 2017, vol. 3. Available from DOI: 10.7717/peerj-cs.111.

11. IYER, R.; YUN, A.; NAGAPPAN, M.; HOEY, J. Effects of Personality Traits on Pull Request Acceptance. *Transactions on Software Engineering*. 2019. Available from DOI: 10.1109/TSE.2019.2960357.

12. GOLZADEH, M.; DECAN, A.; MENS, T. On the Effect of Discussions on Pull Request Decisions. In: *Belgium-Netherlands Software Evolution Workshop*. CEUR Workshop Proceedings, 2019. Available also from: http://ceur-ws.org/Vol-2605/16.pdf.

13. ZOU, W.; XUAN, J.; XIE, X.; CHEN, Z.; XU, B. How does code style inconsistency affect pull request integration? an exploratory study on 117 github projects. *Empirical Software Engineering*. 2019, vol. 24, pp. 3871–3903. Available from DOI: 10.1007/s10664-019-09720-x.

14. LENARDUZZI, V.; NIKKOLA, V.; SAARIMÄKI, N.; TAIBI, D. Does code quality affect pull request acceptance? An empirical study. *Journal of Systems and Software*. 2021, vol. 171, pp. 110806. Available from DOI: 10.1016/j.jss.2020.110806.

15. CHEN, D.; STOLEE, K.; MENZIES, T. Replication Can Improve Prior Results: A GitHub Study of Pull Request Acceptance. In: *International Conference on Program Comprehension*. IEEE, 2019, pp. 179–190. Available from DOI: 10.1109/ICPC.2019.00037.

16. ZHANG, X.; YU, Y.; GOUSIOS, G.; RASTOGI, A. Pull Request Decision Explained: An Empirical Overview. *Computing Research Repository*. 2021. Available from arXiv: 2105.13970.

17. GOUSIOS, G.; PINZGER, M.; DEURSEN, A. An Exploratory Study of the Pull-Based Software Development Model. In: *International Conference on Software Engineering*. ACM, 2014, pp. 345–355. Available from DOI: 10.1145/2568225.2568260.

18. DEY, T.; MOCKUS, A. Which Pull Requests Get Accepted and Why? A study of popular NPM Packages. *Computing Research Repository*. 2020. Available from arXiv: 2003.01153.

19. SOARES, D.; DE LIMA, M.; MURTA, L.; PLASTINO, A. Rejection Factors of Pull Requests Filed by Core Team Developers in Software Projects with High Acceptance Rates. In: *International Conference on Machine Learning and Applications*. IEEE, 2015, pp. 960–965. Available from DOI: 10.1109/ICMLA.2015.41.

20. AZEEM, I.; PENG, Q.; WANG, Q. Pull Request Prioritization Algorithm based on Acceptance and Response Probability. In: *International Conference on Software Quality*. IEEE, 2020, pp. 231–242. Available from DOI: 10.1109/QRS51102.2020.00041.