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. I am gratefully indebted to him for his very valuable comments on this thesis.

Computational resources were supplied by the project "e-Infrastruktura CZ" supported by the Ministry of Education, Youth and Sports of the Czech Republic. This thesis would not have been possible without them.

Abstract

The high code quality makes the software more robust and sustainable and decreases technical debt. This thesis investigates the effect of code quality on the pull request acceptance and on pull request evaluation latency. This will help us understand if the project maintainers pay sufficient attention to the code quality. To unravel the code quality influence, I analyzed one hundred open-source projects from five different programming languages: Python, Java, Kotlin, Haskell, and C/C++. The code quality of the individual pull requests was evaluated using static code analysis. Several machine learning models were utilized to predict the pull request acceptance based on the code quality. The poor code quality seems to have a slight negative impact on the pull request acceptance. However, there seems to be no effect on the time required to close a pull request.

Keywords

code quality, pull request, static code analysis

Contents

1 Introduction

Does code quality influence the acceptance of pull requests? Although many project maintainers view code quality as the most important factor regarding pull request acceptance [1], a recent study by Lenarduzzi et al. shows that the presence of quality flaws in the code does not influence the acceptance or rejection of pull requests [2].

To the best of my knowledge, study by Lenarduzzi et al. is the only one that investigated if quality issues affect the acceptance of pull requests. They analyzed 28 well-known Java projects and applied several statistical techniques to find the relation between code quality and pull request acceptance. The quality of the pull requests was evaluated using the open-source tool called PMD. This tool is able to perform static analysis of the code and detect common programming flaws, such as code-smells, anti-patterns, or code-style violations. Traditional statistical techniques did not find any connection between the code quality and pull request acceptance. Because of that, they trained several machine learning models to predict the acceptance based on the quality issues found in the code. However, machine learning models yielded similar results as traditional techniques.

Unfortunately, the study performed by Lenarduzzi et al., while an extensive study, analyzed only projects written in Java. Moreover, 22 out of 28 analyzed projects were from the Apache Software Foundation. This is the major threat to the generalizability of their findings. To address these shortcomings, I analyzed one hundred projects from five different programming languages — Python, Java, Kotlin, Haskell, and C/C++ (twenty projects per language). For each programming language, a different tool for static analysis was used to evaluate the code quality. I applied similar techniques as Lenarduzzi et al. to analyze the relationship between the code quality and pull request acceptance. Furthermore, the regression models were created to predict the time required to close a pull request using the code quality. Subsequently, the retrieved results were compared between individual languages. The research questions set for this thesis are the following:

RQ_1 Which code issues are typically introduced by the pull requests?

RQ$_2$ Are there some particular quality flaws that affect the acceptance of the pull request?

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

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

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

This thesis first discusses code quality and its link to the pull-based development model (Chapter 2). After that, the various factors that influence the pull request acceptance are mentioned (Chapter 3). The next part of the thesis is dedicated to the research design. At first, I state how the data about the project was retrieved and which methods were used to evaluate the code quality of individual pull requests (Chapter 4). Then, the used statistical methods are introduced (Chapter 5). Furthermore, the retrieved results are evaluated separately for each programming language (Chapter 6). Towards the end, results are compared between languages, and potential threats to validity are discussed. Finally, I summarize my findings and compare them to the findings of Lenarduzzi et al. (Chapter 7). Moreover, the potential extensions of my work are proposed.

2 Code quality in pull-based development

The pull-based development model created novel ways how developers can interact with each other. Instead of pushing code changes (patches) into one central repository, developers can work in a 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 work independently on projects. Furthermore, 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 [3].

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 were created. These platforms add new ways how the developers can interact beyond the basic functionality of Git:

- The forks enable the creation of the server-side copy of the repository.
- Pull requests[1] (on some platforms called merge requests) enables to merge code directly on the platform.
- Users can report issues found in the projects; therefore, the 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 a huge amount of publicly available repositories and GitHub also provides public REST API that can be easily leveraged for data mining.

1. pull request is commonly abbreviated as PR

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

2.1 Measuring code quality

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

To be able to evaluate the software in regard to its quality, there needs to be some way how the code quality can 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 [4].

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

On the other hand, static testing methods suit the analysis of a large number of programs better. Static techniques encompass the usage of formal and informal reviews, walkthroughs, and inspections; however, these techniques are performed by humans and therefore are not viable 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 for the source code's execution.

The ISO/IEC 25010 [5] defines several quality characteristics which can be identified in the software. I will now discuss these characteristics in the context of static analysis:

Performance efficiency evaluates if the application is using the optimal amount of resources. The static analysis can help to create a faster code. For instance, some linters are able to detect constructs/-functions that are ineffective and consume more resources than required.

Usability is the degree to which the software is easy to use. This quality is often evaluated through *usability testing*. On the other hand, there are some properties that can be checked via static analysis, such as proper documentation of public interfaces, which contributes to the application's learnability.

Reliability defines how stable and fault-tolerant the software is. Static analysis can unravel error-prone constructs and multi-threading issues (that negatively influence stability) and ensure that exception handling is properly implemented.

Security is concerned with the confidentiality, integrity, and authenticity of the software. Linters can detect several security-related issues in the source code, such as the use of vulnerable functions or use of the hard-coded values for cryptographic operations.

Maintainability is the ease with which can be application modified. Static analysis can help to ensure that source code is clean and understandable. Source code can be checked if it follows the conventions of the given programming language. For instance, Python has an official style guide for Python code — PEP 8[2]. 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 if the software is properly designed and does not use complicated

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

constructs; for instance linter can detect if some part of the code is redundant, complicated, or too coupled.

Portability is the ability to execute software on multiple platforms. Some linters are capable of detecting functions and data types that are not portable.

However, it is important to note that not all linters have the same capabilities. Issues that can be detected by the given linter heavily depend on the used programming language (some quality issues are language-specific). Which linters were used for the purposes of this thesis is discussed later in the text.

The code issues (the number of their occurrences) identified by linters were used as a metric to evaluate the code quality of the given pull request. The same approach was used by Lenarduzzi et al. [2] during the evaluation of the pull requests code quality.

2.2 GitHub and code quality

GitHub brings many features that may potentially improve code quality. GitHub has a built-in bug tracker which can be used to report issues found in the code. Because the issues can be reported by users outside of the core development team, the code quality issues can be detected earlier and more efficiently. Bug trackers also enable prioritization of issues which helps to decide which problems need attention first.

Moreover, GitHub enables the creation of pull requests — a mechanism by which the developers can propose changes to the code base. When the pull request is submitted, the maintainers of the repository decide if the changes will be applied (merged) or not (rejected). Quality can be one factor that can influence this decision. The versatility of Git enables pull requests to be merged in various ways [3]: through GitHub facilities, using Git merge, or by committing the patch.

One of the pull requests advantages is the integration with the code review functionality. Maintainers of the projects can review the code to improve internal code quality and maintainability.

GitHub provides CI/CD[3] functionality via GitHub Actions[4]. This enables to automatically run static analysis or automated tests whenever some predefined event occurs, such as creating a new pull request. Another possibility is to add a linter directly to the build process and then trigger the build using the GitHub Actions. Trautsch et al. [6] analyzed several open-source projects in regards to the usage of static analysis tools. They found out that incorporating a static analysis tool in a build process reduces the defect density.

3. continuous integration/continuous delivery
4. `https://github.com/features/actions`

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. [1] 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. [7] 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 [8]. 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 can approach the pull request acceptance in a different manner, three main governance styles can be identified — protective, equitable, and lenient. The 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 [9]. Each style focuses on different aspects of PR. Tsay et al. [10] 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 of being accepted than PRs containing the code written in Go or Scala [11].

Furthermore, older projects and projects with a large team have a significantly lower acceptance rate [10].

The popularity of the project also influences the acceptance rate — projects with more stars have more rejected PRs [10].

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 [10].

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% [10].

Moreover, users that contributed to a larger number of projects have a higher chance that their PR will be accepted [12]. The acceptance of the new pull request also correlates with the acceptance of other older pull requests created by the same submitter [13] [14]. Furthermore, the first pull requests of users are more likely to be rejected [15].

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 [16].

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 [17].

3.3 Pull request level

The *pull request level* is interested in the data about 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.

One of the factors that negatively influence the acceptance rate is the number of commits in the pull request. The high number of commits decreases the probability of acceptance. On the other hand, PRs with only one commit are exceptions — they have a smaller chance of being accepted than pull requests which contain two commits [12].

Another observation is that more discussed PRs have a smaller chance of being accepted [10]. Another study did not find a large difference between accepted and rejected PRs based on the number of comments but found that discussions in rejected PRs have a longer duration [18]. Moreover, the increasing number of changed lines decreases the likelihood of PR acceptance [10].

The code quality is an essential factor on the *pull request level*, and it is this study's main interest. The code quality as the acceptance factor is examined in the following subchapter.

Code quality

One of the instruments that ensure that the code has high quality is testing. Proper testing is a crucial part of every project. Testing plays a significant role in discovering bugs and therefore leads to higher code quality. One study found that PRs, including more tests, have a higher chance of being accepted [10]. However, another study yields no relation between acceptance and test inclusion [3].

Another factor that is closely tied to code quality is the code style. Proper and consistent code style increases the maintainability of the software. The code style inconsistency has a small (but not negligible) negative effect on acceptance. PRs with larger code style inconsistency (with the codebase) have a smaller chance of being accepted. Code style inconsistency also negatively influences the time required to close a PR [19].

Although many integrators view code quality as the most important factor regarding PR acceptance [1], to the best of my knowledge, only one study [2] was performed to discover whether there is a con-

nection between the PR's acceptance and the quality flaws found in the code (taking into account more complicated aspects than code style or test inclusion).

Lenarduzzi et al. [2] analyzed 28 open-source projects. The results show that there is no significant connection between code quality and PR acceptance. The key difference (from my thesis) is that they analyzed only projects written in Java. Furthermore, my thesis investigates the connection between the time to close a PR and the PR quality. Further comparison is at the end of the thesis.

4 Data mining

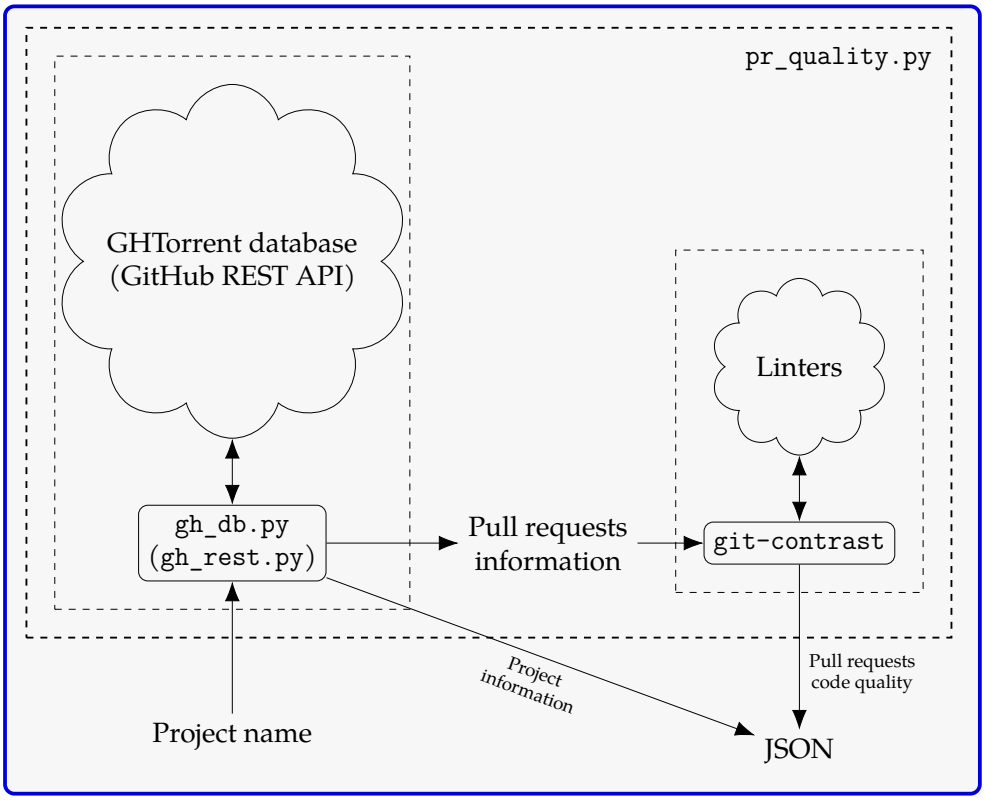

Figure 4.1: The `pr_quality.py` workflow

Information about the pull requests is retrieved using the `pr_quality.py` script. This script takes the names of the projects that will be analyzed as the input, and it outputs the JSON files containing information about the projects and their code quality (Figure 4.1). The script needs to retrieve the metadata for each project and its pull requests. There are two possible sources that can be used: GitHub REST API and the GHTorrent database. Which source will be used can be specified by passing an argument to the tool. Metadata are then used to determine which objects need to be fetched from the GitHub to perform the code quality analysis. The analysis of the pull request itself is performed by an external tool called `git-contrast`.

The `gh_db.py` is a script responsible for querying the GHTorrent database in order to obtain data about the projects. The GHTorrent database [20] is an offline mirror of data offered through the GitHub REST API. `gh_db.py` returns a JSON file with the information about the project, such as the number of stars, number of contributors, or information about pull requests and their commits.

An alternative script that can be used by `pr_qality.py` is `gh_rest.py`. This script uses the GitHub REST API directly. The advantage of this script is that it can retrieve the newest data from GitHub. Unfortunately, the REST API is limited by the number of requests per hour. Because of that, the `gh_rest.py` is programmed to retrieve only a subset of data that are obtained by `gh_db.py` (data not crucial for the analysis are omitted).

However, GitHub lacks information about the code quality of the pull requests. This is where the `git-contrast` comes into play. `git-contrast` is the command-line application that analyzes the code quality of the given pull request using the external linters. This application is further discussed in the following sections.

4.1 GitHub metadata

As stated before, the scripts `gh_db.py` and `gh_rest.py` are used to retrieve data from GitHub. GitHub can be leveraged to obtain many interesting metadata, which can possibly influence the acceptance of pull requests. All the metadata that are obtained using the scripts are listed in Table 4.1.

Metadata like "Number of stars" or "Time opened" are required for the statistical analysis. Others are not meant to be used as a part of the analysis itself but are kept here for better orientation, and some of them are needed for the `git-contrast` tool, such as "Head commit", "Base commit", etc.

Table 4.1: Data retrieved from GitHub

Level	Metadata	`gh_db.py`	`gh_rest.py`
Repository level	Project name	✓	✓
	Programming language	✓	✓
	Time of creation	✓	✓
	Number of forks	✓	✓
	Number of commits	✓	✗
	Number of project members	✓	✗
	Number of stars	✓	✓
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	✓	✗

4.2 Evaluating code quality

`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. The `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. The number of found code quality issues is then written to the standard output.

To measure the change in the quality of the pull request, the `git-contrast` is run on the "head commit" and the "base commit" of the given pull request. The `git-contrast` supports several linters; which linter will be used is determined by the file extension of the tested file (Table 4.2).

Table 4.2: Linters supported by `git-contrast`

Linter	Version	Programming language	File extensions
Pylint	2.12.2	Python	`.py`
PMD	6.42.0	Java	`.java`
ktlint	0.43.2	Kotlin	`.kt` and `.kts`
HLint	3.2.8	Haskell	`.hs`
flawfinder	2.0.19	C/C++	`.c`, `.cpp` and `.h`

The most problematic was to statically analyze the C/C++ source files because some linters also need the information on how the source

code should be compiled. I tested the OCLint and Cppcheck linters but without success. The compilation flags cannot always be automatically determined from the makefiles. Because of that, I settled on using the flawfinder, which performs a simpler analysis and does not require compilation flags.

The following linters are supported by `git-contrast`:

Pylint Python linter that is able to detect programming errors and helps enforce coding standards[1]. Issues are divided into the following categories: conventions, code smells, warnings (Python-specific problems), and errors.

PMD Linter that is able to discover common programming flaws. It is mainly concerned with Java and Apex programming languages. PMD is extensible but also provides many predefined rulesets: "Best Practices", "Code style", "Security"... All Java rule sets available in the basic installation were used to evaluate code quality.

ktlint Simple static analyzer focused on the code clarity and community conventions[2]. This linter uses only a small set of carefully selected rules.

HLint Tool for suggesting possible improvements to Haskell code. Every hint has one of the following severity levels: error, warning, and suggestion.

flawfinder A simple program that examines C/C++ code and searches for potentially dangerous functions. This is done using the built-in database of functions with well-known problems. Linter uses the following risk levels: note, warning, and error.

4.3 Projects selection

In total, 100 projects were selected written in five different programming languages (20 projects for each language). The analyzed GitHub projects were selected based on the following criteria:

1. `https://peps.python.org/pep-0008/`
2. `https://kotlinlang.org/docs/coding-conventions.html`

- The primary programming language is Python, Java, Kotlin, Haskell, or C/C++.
- The project is popular — it is in the top 150 most favorite projects written in the given language. One of the reasons to analyze popular projects is the fact that popularity influences acceptance [10]. Popular projects also usually contain a high number of pull requests. Two different lists of popular projects were used: projects sorted by the number of stars using the GHTorrent database (data from 1st June 2019) and the list from GitHub[3] (data from 1st January 2022).
- The project contains at least 200 pull requests that are suitable for analysis. This means that PR needs to contain at least one file written in the primary language and the data about PR needs to be publicly available.
- The project is using GitHub to merge pull requests (for most of the pull requests).
- The project is a library, program, or collection of programs. Repositories whose primary purpose is to store configuration files, documentation, books, etc., were ignored.

3. `https://github.com/EvanLi/Github-Ranking`

5 Data analysis

In this chapter, I am explaining which statistical methods were chosen in order to answer the research questions. RQ_1–RQ_4 were analyzed separately for each programming language; therefore, also the techniques that will be discussed were applied separately. Only the last research question discuss multiple languages at the same time and compares results retrieved from the individual analysis of each language.

5.1 RQ_1: Which code issues are typically introduced by the pull requests?

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

I also summarized all pull requests regardless of their project. I computed the average number of introduced issues, fixed issues, etc. Then I created a 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 requests introduced/fixed this issue, how many times this issue occurred in some pull request, etc. I created multiple lists of issues sorted by various parameters. I sorted issues by the number of rejected/accepted PRs that fixed/introduced them. I also listed issues and the percentage of PRs that changed their quality. I examined the issues that were fixed in a larger number of PRs than introduced. Then I created a scatter plot that shows which issue category is the most common.

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

5.2 RQ_2: Are there some particular quality flaws that affect the acceptance of the pull request?

In order to discover issues that affect the acceptance of pull requests most, the classification models were created. The aim of these models is to classify pull requests into two groups (accepted PRs and rejected PRs) by using the information about the quality change in the given pull request. Multiple classification algorithms were used[1]:

LogisticRegression [21] Despite its name, logistic regression is a linear model used for classification. It uses a so-called *logistic function* that turns the inputs (code quality issues) into the probability of the dependent variable (PR acceptance) being 1 (PR is accepted).

DecisionTrees [22] This algorithm constructs the tree where leaves represent the different classes (PR accepted/rejected), and inner nodes represent the so-called *split criterion* — the condition (or predicate) on single/multiple attributes (code quality issues). The *split criterion* defines to which subtree given input (pull request) belongs.

Bagging [23] The Bagging algorithm is trying to predict the data class (PR being rejected/accepted) using multiple different classifiers. It uses bootstrapping[2] to construct the different data sets for each classifier. The outputs from these classifiers are then aggregated to form the final prediction.

RandomForest [24] This classifier leverages the bagging method in order to create the forest of uncorrelated decision trees (to avoid bias and overfitting). Unlike the decision trees, the RandomForest uses only a subset of features (code quality issues) to generate the decision tree (this ensures the low correlation between the trees).

ExtraTrees [25] ExtraTrees is a classifier similar to RandomForest. The main difference is that the ExtraTrees algorithm generates *split criterions* using randomization. Another key difference is that

1. `https://scikit-learn.org/stable/modules/classes.html`
2. random sampling with replacement

ExtraTrees uses whole original sample for each tree (instead of bootstrapping).

AdaBoost [26] The AdaBoost is another algorithm that leverages multiple weak classifiers (usually DecisionTrees with only one *split criterion*) to predict the final result. It begins by fitting a classifier on the original dataset. Each subsequent classifier is improved using the results from the previous one (incorrectly classified pull requests have a higher chance of being selected in the next classifier).

GradientBoost [27] The GradientBoost algorithm is similar to the AdaBoost. It is also using multiple weak classifiers, and they are trained one by one. However, instead of improving the subsequent classifier by changing the training dataset distribution, the GradientBoost algorithm trains the classifiers using the residual errors of predecessors. Furthermore, the GradientBoost works with larger trees than AdaBoost.

XGBoost [28] XGBoost is a popular variant of gradient boosting. It is designed to be fast and efficient. It can generate multiple tree nodes in parallel. Furthermore, *regularization* is used to prevent overfitting.

Each of those algorithms was run on three different datasets:

- a dataset with quality change
- a dataset containing only introduced issues
- a dataset with only fixed issues

In the first dataset, the quality change for some issues was represented by the integer, and this integer was negative if the issue was fixed in the PR and positive if the issue was introduced. The other datasets were created by filtering positive/negative values from the first dataset. Running the classification algorithms on the dataset with only fixed issues can help to understand if the improvement in code quality can also influence the acceptance.

In order to recognize issues that have some effect on the PR acceptance, the *drop-column importance* mechanism[3] was used. This mechanism is resource-intense (requires a lot of computational power) but is usually more reliable than the classic importance mechanisms.

The dataset was split into five parts to better evaluate the model accuracy (5-fold cross-validation). Each model was then trained five times — a distinct dataset was used for training and for validation. Several metrics (precision, recall, AUROC, F-measure…) were used to evaluate the reliability of each model. Afterward, the average metrics over all folds were computed.

The same technique was used by Lenarduzzi et al. [2]. The script they provided was used to run the classification algorithms. It was only slightly modified to improve the user interface. Furthermore, the option to filter only fixed/introduced issues was added.

5.3 RQ_3: Is there a relationship between the source code quality and the pull request acceptance?

At first, the PCA (principal component analysis) scatter plot was created to visualize the difference between accepted and rejected pull requests.

The impact of the presence of some code issue in the PR on the PR acceptance was determined using the χ^2 test. In order to perform this test, the dataset was transformed into a *contingency table*. This table (2×2) contained the number of accepted/rejected PRs with/without a code quality issue. After that, the χ^2 test of independence was performed on the *contingency table*. The *significance level* was set to $\alpha = 0.05$. However, relying only on statistical significance can be misleading because it is affected by sample size. To understand the practical significance of the test (*effect size*), the Cramer's V denoted as ϕ_c was also computed. The Cramer's V ranges between 0 (no association) and 1 (complete association).

Pull request that adds or removes some files greatly influences code quality. If the number of removed/added files has a large impact on PR acceptance (regardless of code quality), then it can be a large threat to the validity of the independence test. The pull request acceptance

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

can also be influenced by the quality of files which were not linted (were written in non-primary language). To eliminate the risk that the test was influenced, the same test was performed on pull requests that only modified some source files, and these files were written in the primary language.

Moreover, the χ^2 test was performed independently for each issue category to understand if there are some issue categories that have a stronger influence on the quality.

The test was also computed for each project separately. Unluckily, there are some projects that contain an insufficient number of pull requests. According to Cochran [29], all expected counts should be ten or greater. Therefore, the tests were performed only on some projects (that have a sufficient number of expected counts).

It is important to note that p-values were not adjusted in any way.

The metrics obtained from classification algorithms were also used to determine if the code quality has some impact on PR acceptance.

5.4 RQ_4: Does code quality influence the time required to close a pull request?

In order to find the possible link between the code quality and the time it takes to close a PR, 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. Afterward, the model was used to predict the time based on the data from the 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 and used to evaluate the models.

Following regressors were used[4]:

LinearRegression [30] Linear regression is a commonly used type of predictive model. It is used for modeling the linear relationship between explanatory variables (code quality issues) and a scalar

4. `https://scikit-learn.org/stable/modules/linear_model.html`

response (time to close a PR). The model that minimizes the residual sum of squares is selected.

ElasticNet [31] ElasticNet is an extension of linear regression. It is adding L_1 (lasso regression) and L_2 (ridge regression) penalties in order to make the linear model more robust. The problem with the classic linear regression is that the estimated coefficients can be too high due to overfitting. Because of that, the model parameters are added to the *loss function*[5] as a penalty.

Some of the already discussed methods used for classification were also used for regression. Following methods were used for both classification and regression: **DecisionTree**, **RandomForest**, **AdaBoost**, **Bagging**, and **GradientBoost**.

5.5 RQ_5: Is code quality impact higher in projects that are using some particular programming language?

The RQ_3 discusses the impact of code quality on individual programming languages. The findings from the RQ_3 for each language are compared in the RQ_5. This comparison is a complicated task because each language has different characteristics, and a different linter was used to measure its code quality.

The results from χ^2 tests were compared to identify the possible difference between the languages (in terms of code quality). The metrics retrieved from classification models were also compared. Finally, the code quality effect on the time to close a PR was compared between the languages (using the metrics from regressors).

5. a function that is minimized during the regression

6 Evaluation

This chapter is dedicated to the findings from my research. The first five subchapters focus on individual programming languages — Python, Java, Kotlin, Haskell, and C/C++. In these subchapters, I am giving the answers to the first four research questions. The last research question (RQ_5) is answered afterward. At the end of this chapter, I am discussing possible threats to validity that could eventually influence the outcomes of my study.

6.1 Python

In order to analyze the influence of code quality on the pull request acceptance, 20 projects from the Python ecosystem were selected. In total, 9452 pull requests were analyzed, and 73 % of these PRs were accepted. As shown in Figure 6.1, pull requests were more accepted in less popular projects.

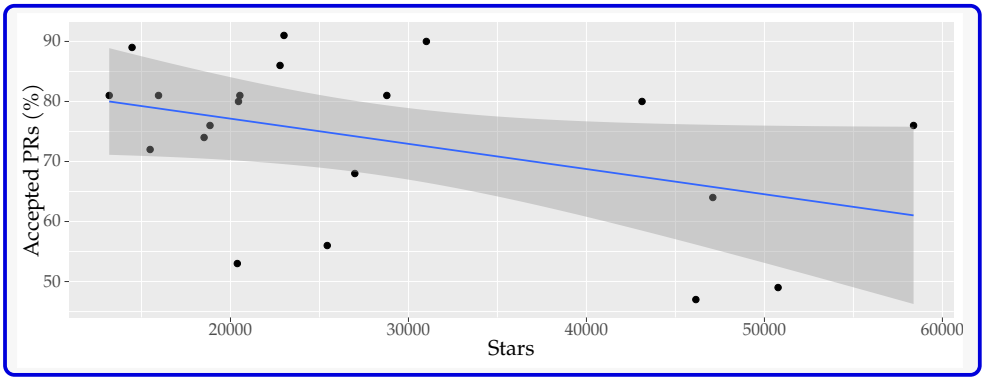

Figure 6.1: Stars and pull request acceptance

On average, one pull request introduced 5.36 issues and fixed 2.44 issues (see Figure 6.2); an 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.

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

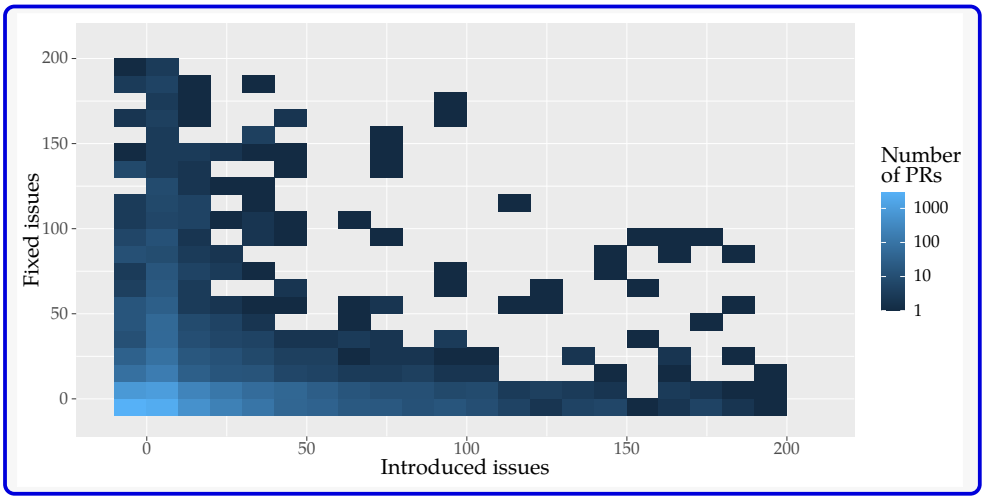

Figure 6.2: Pull requests and quality

The conventions dominated the list of issues that were fixed/introduced in the largest number of pull requests (Figure 6.3). The convention that was fixed/introduced in the largest number of pull requests is `missing-function-docstring` (in 37 % of PRs); conventions `invalid-name`, `line-too-long` and `consider-using-f-string` were fixed/introduced in over 20 % of pull requests. There were 15 issues that were fixed/introduced in more than 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 nine issues that were present in the analyzed pull requests but did not influence their quality (the 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 many false positives that arise due to linting in the isolated environment. Sixty issues were fixed in more PRs than they were introduced. They are 24 more PRs that fixed the warning `super-init-not-called` than the PRs that introduced it.

The most important Pylint issue in regards to the PR acceptance is the `syntax-error` (see Figure 6.4). XGBoost classifier gives this error the 1.2 % importance. However, other classifiers consider this error

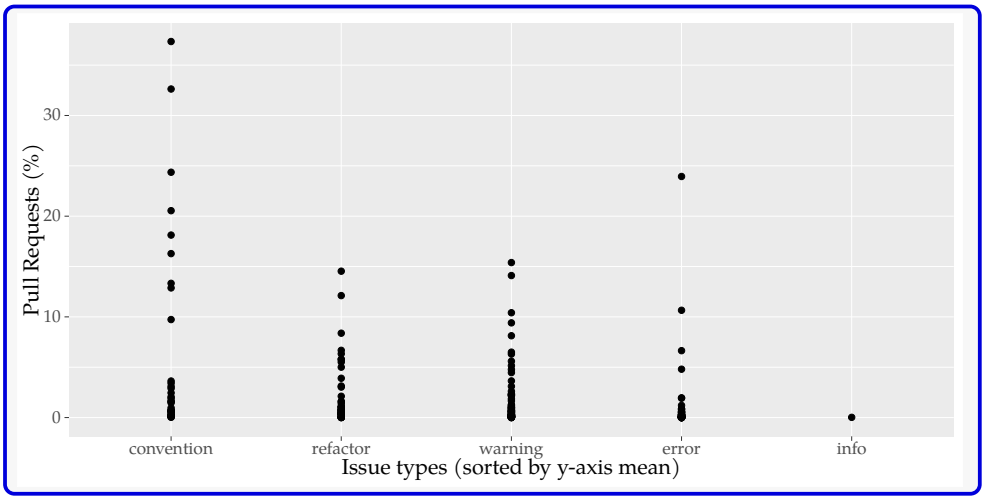

Figure 6.3: Pylint issues and % of PRs which fixed/introduced them

less important. On average importance of the `syntax-error` is only 0.3 %. The syntax error was introduced in 17 projects. On average, rejected pull request introduced `0.027` syntax errors, and the average accepted pull request even fixed `0.001` syntax errors.

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 the top 10 in both lists: `syntax-error`, `unused-variable` and `unused-import`. The `syntax-error` is considered the most important issue by both methods.

When only the information about fixed issues is used, 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). 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 a χ^2 test of independence. As can be seen in Figure 6.6, the observed number of rejected pull requests which contained some defected is higher than

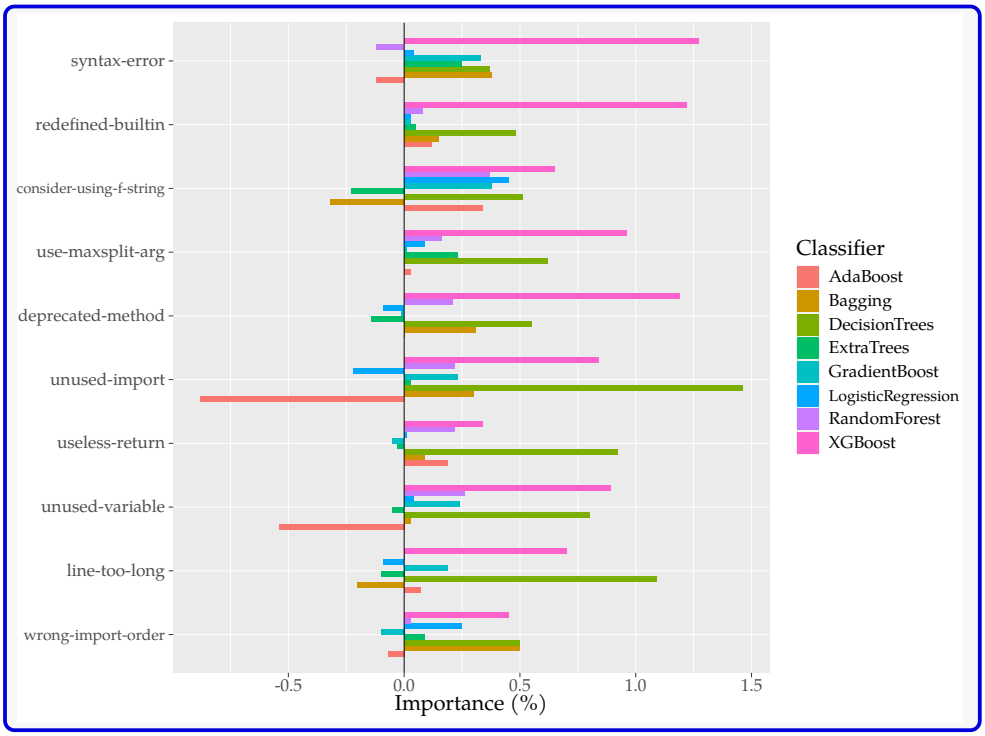

Figure 6.4: Ten most important Pylint issues

expected. For χ^2 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 between issue presence and acceptance is weak. This conclusion also supports the fact that AUROC for trained classification models is only slightly over 0.5. The average AUC for all models is 0.534.

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

Almost identical results were obtained when the χ^2 test was performed separately for each issue category.

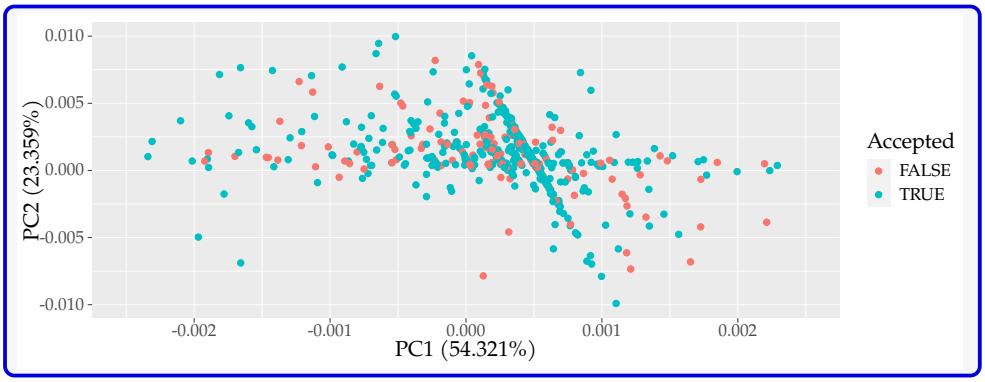

Figure 6.5: PCA scatter plot

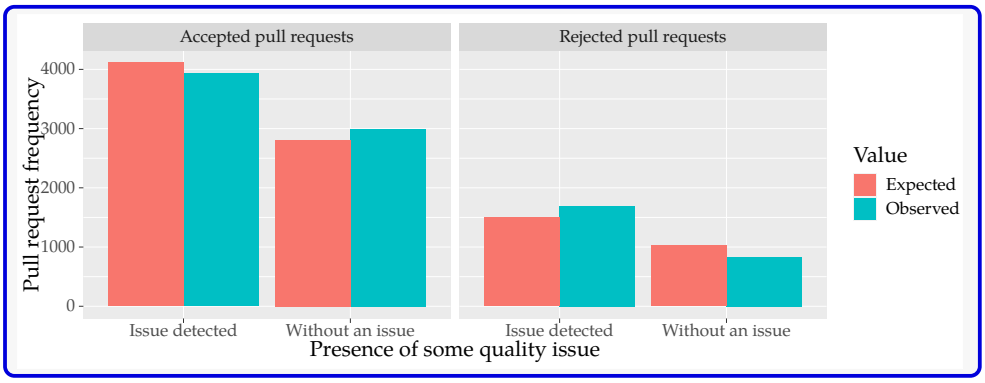

Figure 6.6: Relationship between presence of issue and PR acceptance

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, the presence of some code quality issue does not seem 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 a 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 73 % of these pull requests were accepted. On average, the one pull request introduced 20 new PMD issues but, at the same time, also fixed 18 other issues.

Like in the Python projects, the pull request from the less popular project were more likely to be accepted than pull requests from more popular projects (see Figure 6.7).

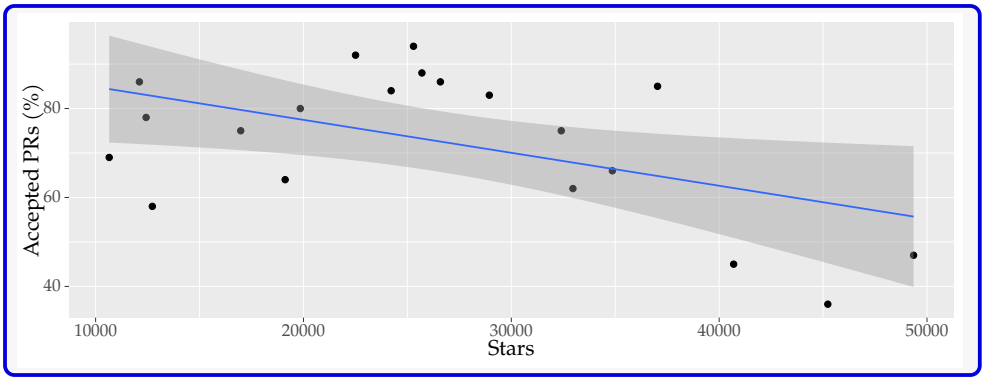

Figure 6.7: Stars and pull request acceptance

As shown in Figure 6.8, only 1366 pull requests (from the total of 8887 pull requests) did not change the quality of the source code (did not fix nor introduce some PMD issues). 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 a million code-style issues.

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 were introduced in more than 3000 pull requests. Similarly, the list of issues that were fixed in the largest number of the pull request is dominated by the very same issues.

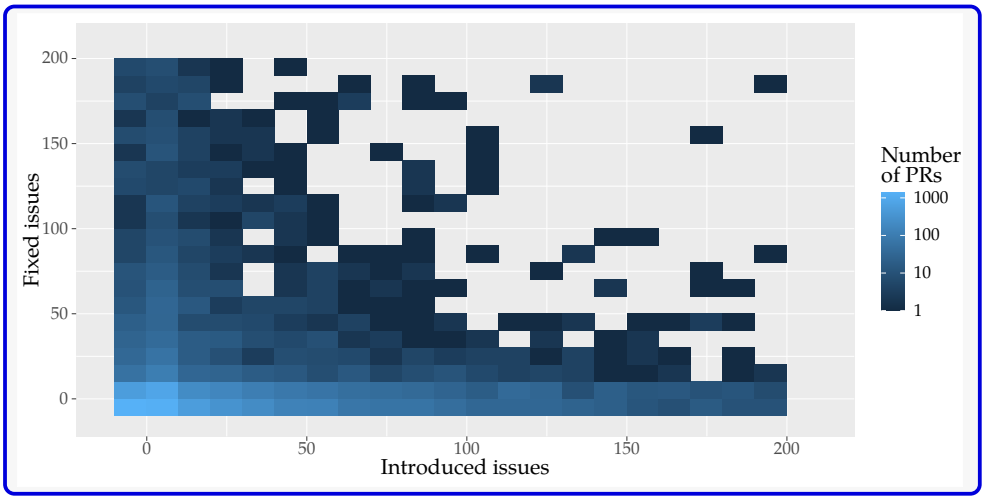

Figure 6.8: Pull requests and quality

As can be seen in Figure 6.9, the documentation issues tend to appear in a large number of pull requests (24 % on average). Moreover, the typical code style issue appeared in 11 % of pull requests. On the other end of the spectrum, an average issue indicating an error-prone construct is present in only two percent of pull requests.

The most important PMD issue is `JUnitAssertionsShouldIncludeMessage` (Figure 6.10). 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 an average accepted pull request. I suspect that the pull requests that are adding a larger number of tests to the codebase have a higher probability of being accepted. At the same time, these pull requests also have a higher probability of introducing the `JUnitAssertionsShouldIncludeMessage`. This can be the reason why this issue has the largest importance. This also supports the study that shows that the acceptance likelihood is increased by 17.1 % when tests are included [10]. However, another performed study indicates that the presence of test code does not influence PR acceptance [3].

The PCA scatter plot was created to understand the differences between accepted and reject pull requests (Figure 6.11). However, there is no visible difference.

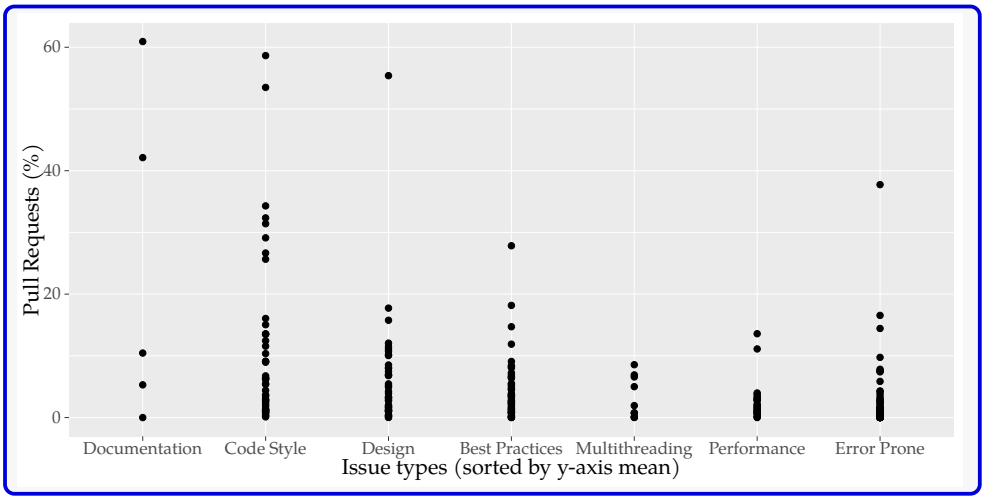

Figure 6.9: PMD issues and % of PRs which fixed/introduced them

To understand the impact on pull request acceptance when a quality issue is introduced, the χ^2 test was performed (see Figure 6.12). 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 the source code of the main language were considered and also when the test was performed individually for each issue category.

17 out of the 20 Java projects contained a sufficient number of pull requests to perform the χ^2 tests. In nine of them, the code quality and acceptance are not independent. Unexpectedly, in one of the projects (`alibaba/fastjson`) the presence of an issue has a small positive effect on the acceptance.

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

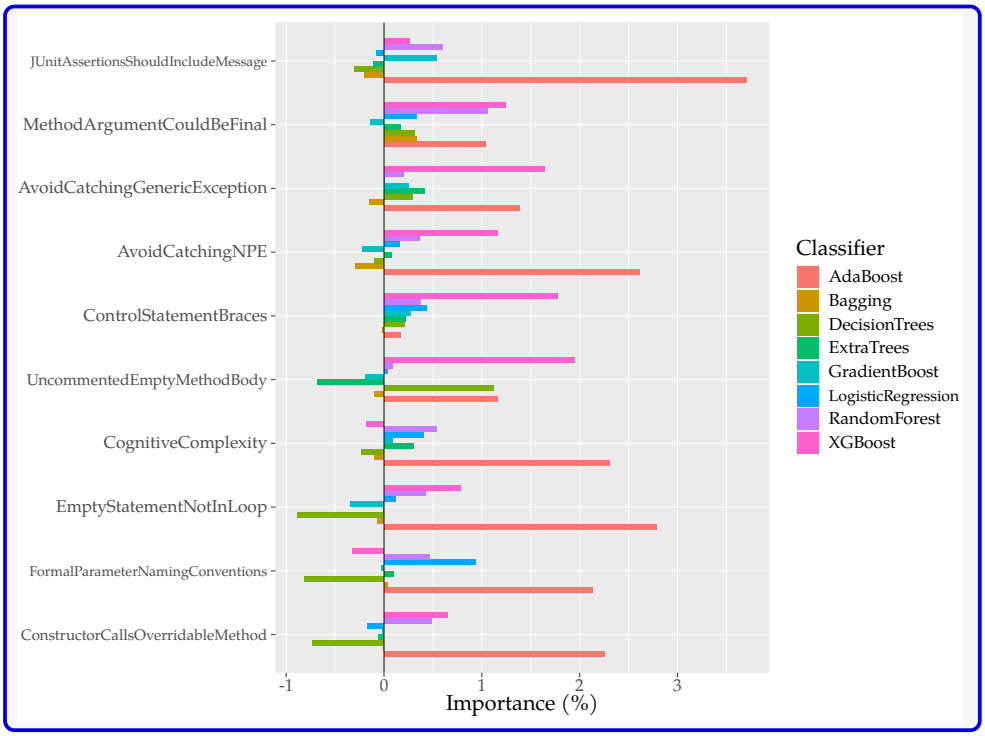

Figure 6.10: Ten most important PMD issues

absolute error (MAE). The average MAE value for all of the models is 3934338 $\approx$ 46 days and 87 % of all analyzed Java pull requests were 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 do not seem to have an effect on the time to close a pull request.

6.3 Kotlin

The 20 projects were also selected from the Kotlin ecosystem. The average analyzed pull request was from a 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

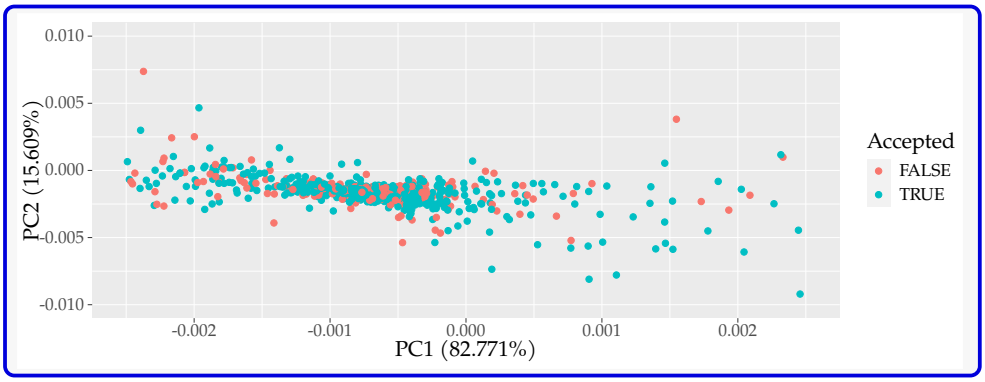

Figure 6.11: PCA scatter plot

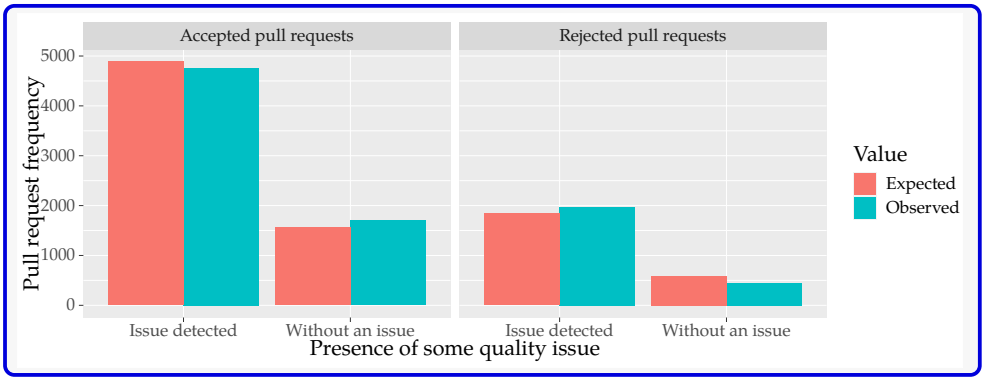

Figure 6.12: Relationship between presence of issue and PR acceptance

more pull requests can also be observed in the Kotlin community (Figure 6.13).

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

The `indent` is the issue that was introduced in the largest number of pull requests (2598). It is the only issue that was introduced in more than a thousand pull requests. It is also the issue that was fixed in the largest number of pull requests. The official Kotlin convention is to use the four spaces for indentation[1], and the `indent` issue signifies that

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

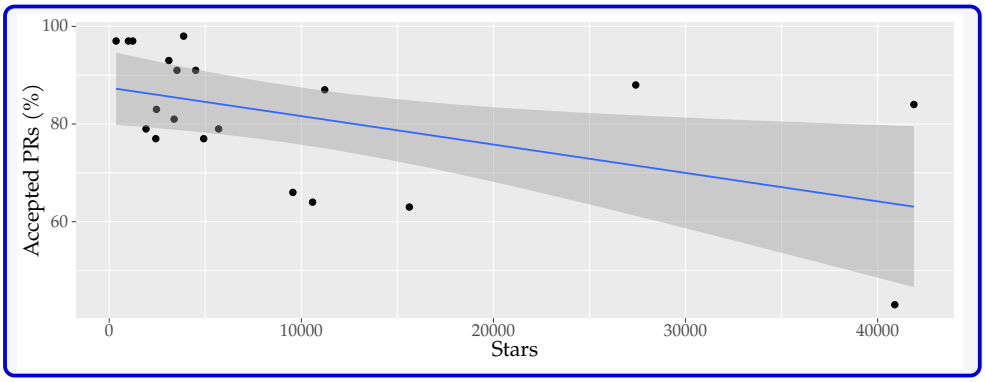

Figure 6.13: Stars and pull request acceptance

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 do 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.

- **TODO**: reference figure "Pull requests and quality"

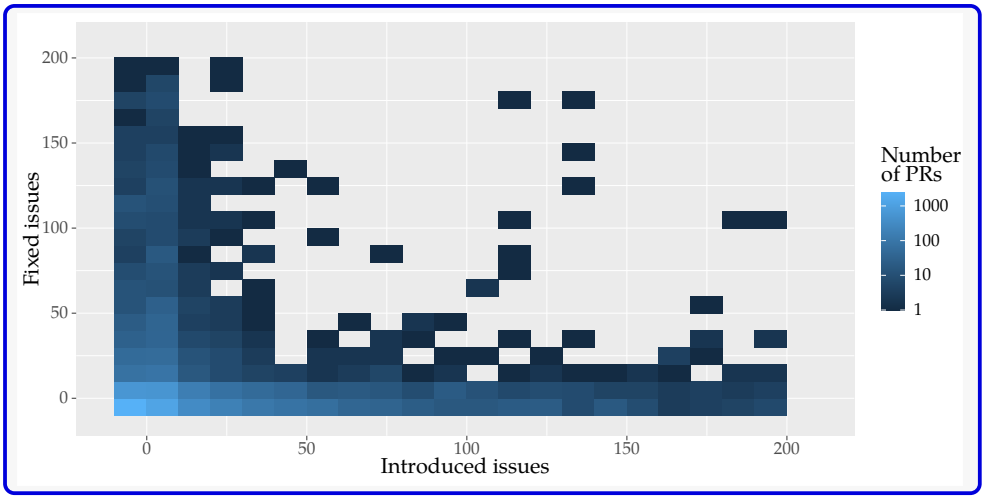

Figure 6.14: Pull requests and quality

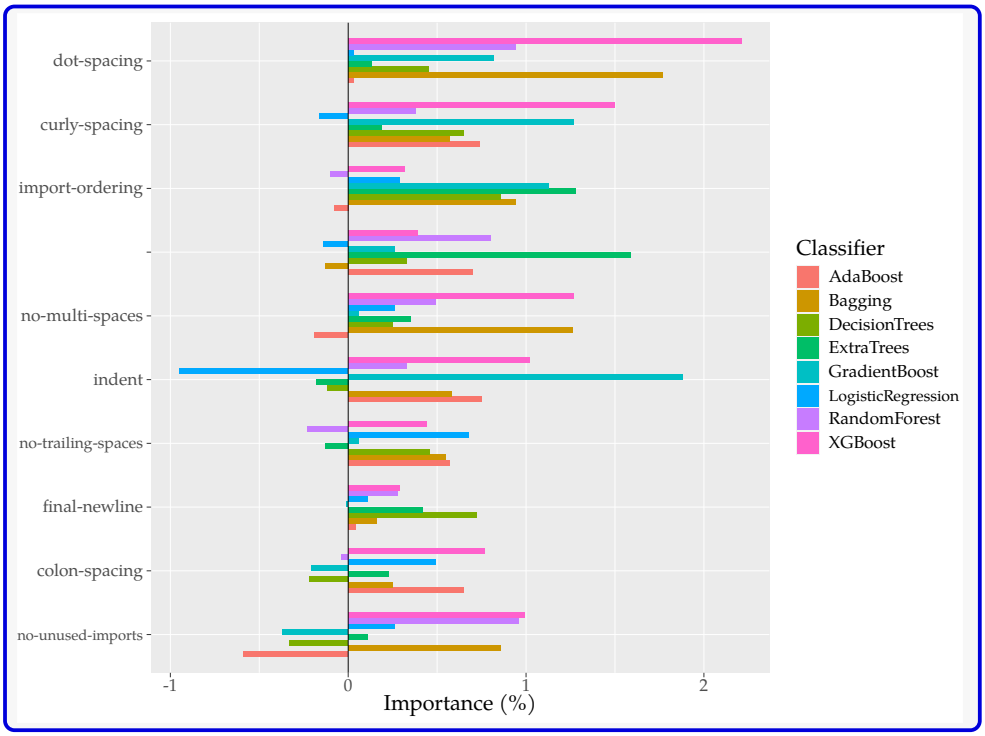

Figure 6.15: Ten most important ktlint issues

As shown in Figure 6.15,the issue with the highest importance average is `dot-spacing`. The Bagging classifier gives 1.7 % importance to this issue. The importance obtained from other classifiers is smaller — the average importance is 0.8 %. However, this issue was introduced only in 18 PRs (13 times in the rejected pull request). Furthermore, seven accepted and seven rejected pull requests fixed this issue. Therefore the impact of this issue is disputable.

It is worth mentioning that fourth most important issue does not have a name (given by *ktlint*). This issue usually indicates an 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 a false positive.

The PCA scatter plot was also created for the Kotlin programming language (Figure 6.16). The first 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:

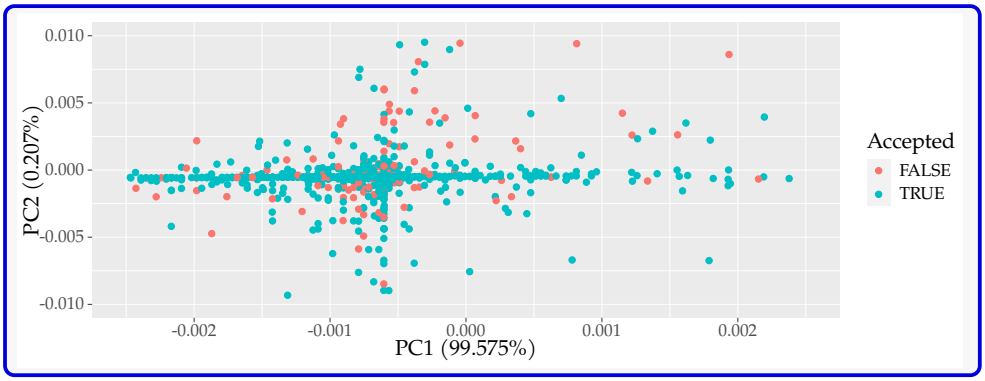

Figure 6.16: PCA scatter plot

To understand the link between acceptance and the introduction of some quality issue, I performed the χ^2 test on Kotlin dataset (see Figure 6.17). 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. 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 a sufficient number of pull requests to evaluate the χ^2 test. There are four projects where the presence of some issue has a 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, I applied several regression techniques also to the Kotlin dataset. For linear regression, $R^2 = 0.164$, therefore the trained model is able to explain 16 % of the variance in the time to

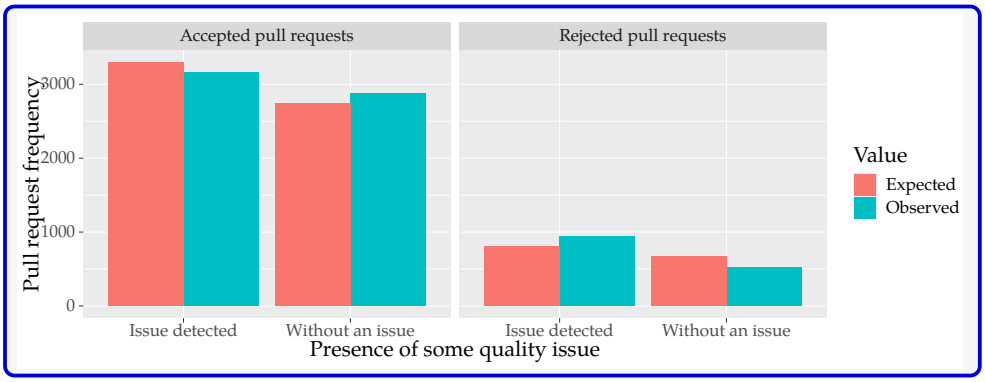

Figure 6.17: Relationship between presence of issue and PR acceptance

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 requests 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 was analyzed. The 18 out of 20 selected Haskell projects have under the 5000 stars. There are only two exceptions: PureScript with 7632 stars and Pandoc, which 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 the number of stars and acceptance in the selected projects. When the outliers are filtered, the trend tends to be the opposite of previous languages: more accepted are pull requests of projects with more stars (Figure 6.18). However, only 20 projects are not sufficient to make such conclusions about the whole population of Haskell projects.

The 6949 pull requests were analyzed. Interestingly, in over 60 % of pull requests, no change in the code quality was detected (see Figure 6.19). Moreover, the *HLint* is able to recognize a large number of different issues (321 issue types were detected in selected pull requests). On the other hand, some issues were counted twice because

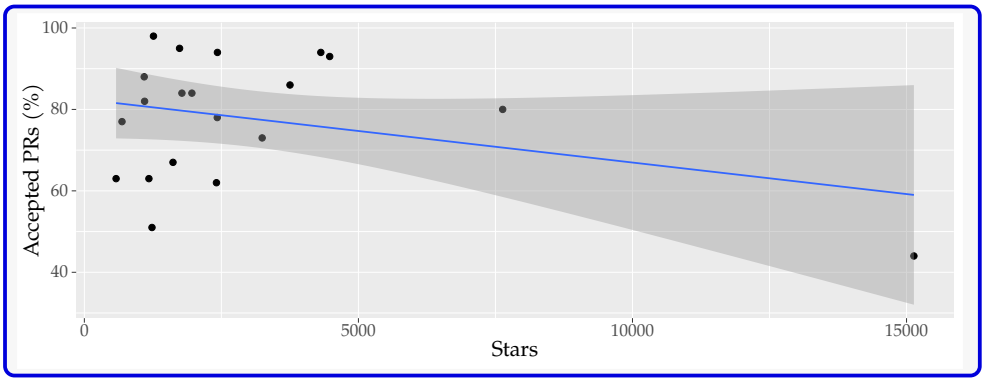

Figure 6.18: Stars and pull request acceptance

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

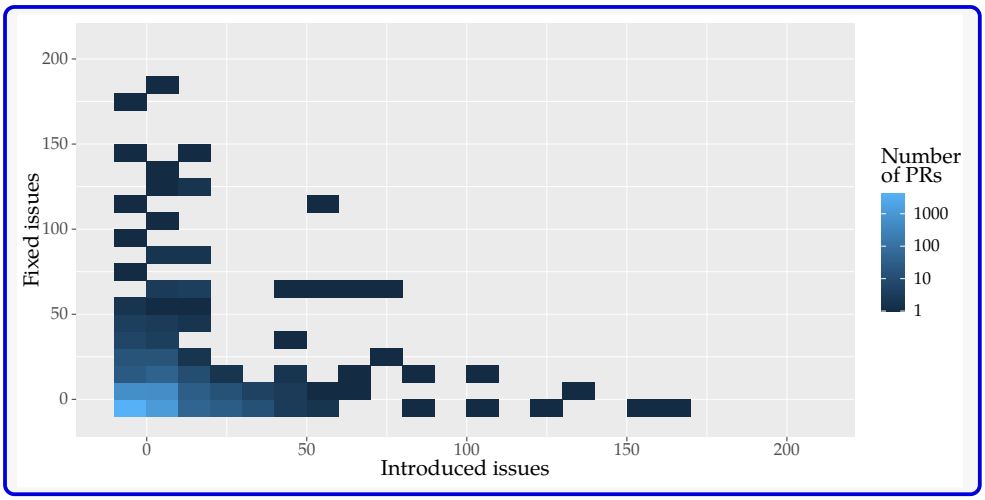

Figure 6.19: Pull requests and quality

Seventy-eight percent of pull requests were 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 (Figure 6.20). 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; and another 105 issue types were detected in the analyzed code, but no PR introduced any of those issues.

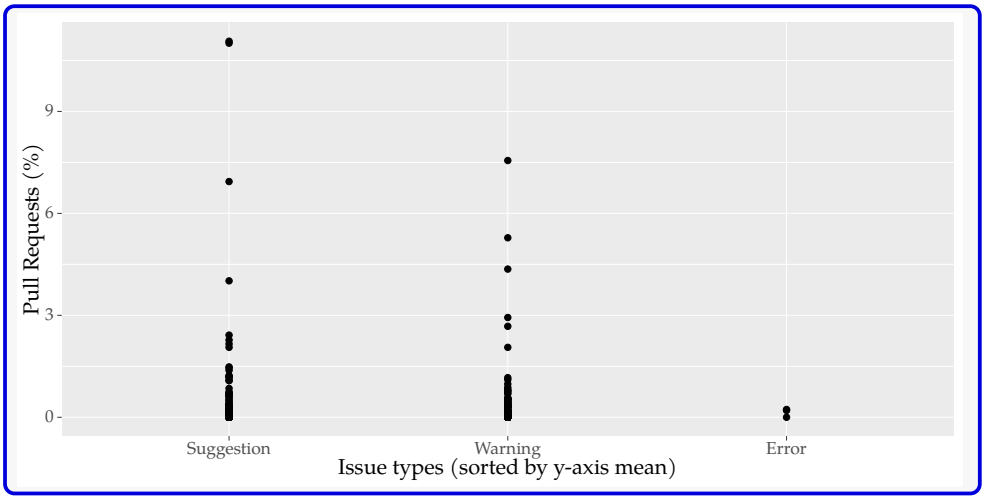

Figure 6.20: HLint issues and % of PRs which fixed/introduced them

As can be seen in Figure 6.21, the most important Haskell issue is the 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. When only introduced issues were taken into account, the most important issue is `Move-brackets-to-avoid-$` (suggestion). The AdaBoost classifier gives this issue 1 % importance, although the average importance is only 0.4 %.

In the 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 (Figure 6.22). Similar to the results in already analyzed languages,

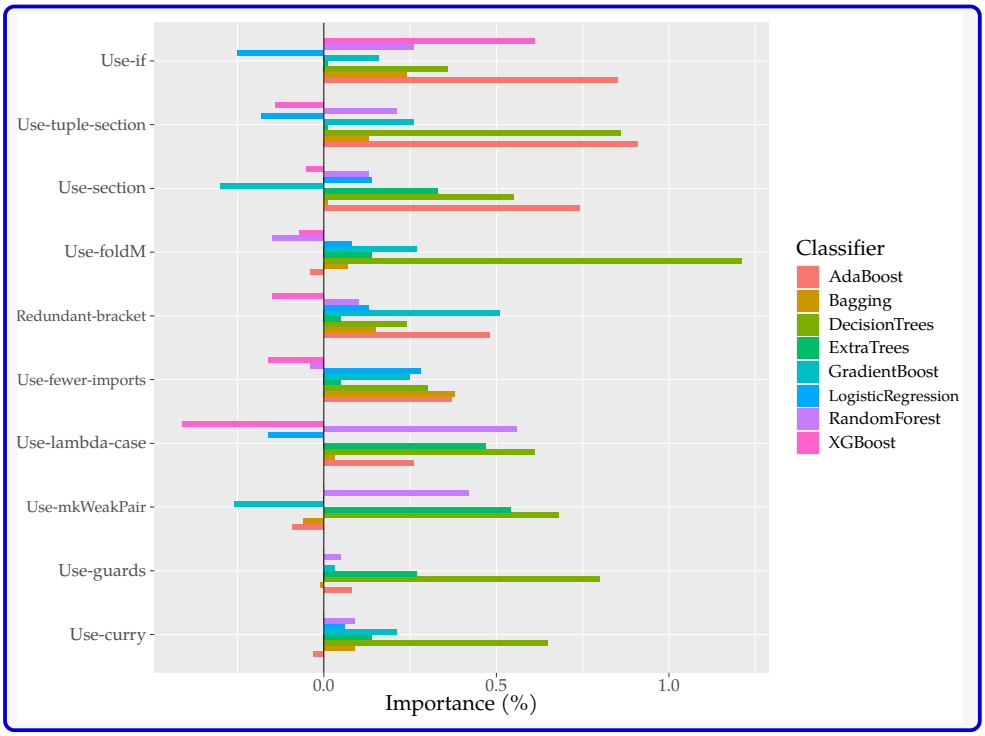

Figure 6.21: Ten most important HLint issues

there is no apparent difference between accepted and rejected pull requests.

For the χ^2 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 PRs has only a small negative impact on the acceptance of the pull request (see Figure 6.23). Similar results were obtained when only the pull requests that contain exclusively some modified code were considered. Furthermore, tests for the individual issue types also yielded similar results. Unfortunately, there is only a small number of pull requests that introduced some errors; therefore the χ^2 test cannot be performed on this issue category. The average AUC computed for ROC curves is around 50 — the classification algorithms were unable to distinguish between the accepted and rejected PRs using the code quality. The 13 projects contain a sufficient number of pull requests; the acceptance and the issue presence are not independent only in four of them (there,

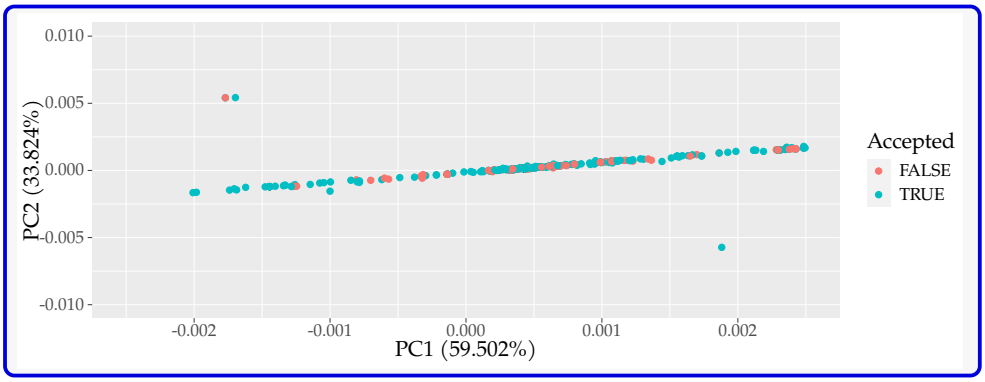

Figure 6.22: PCA scatter plot

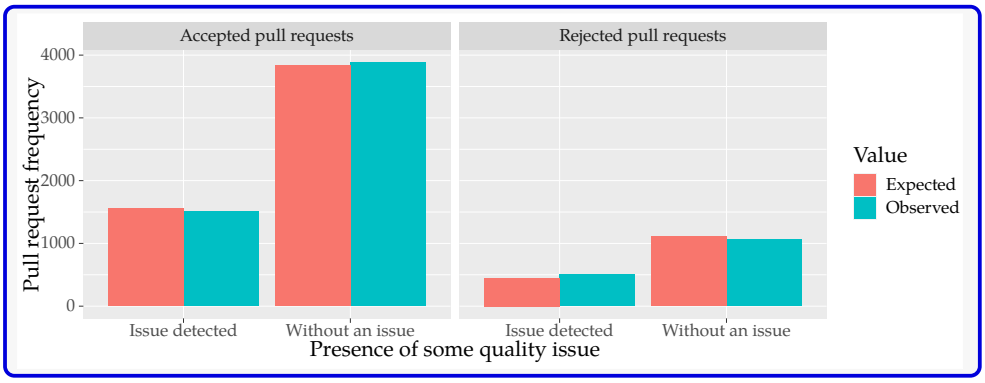

Figure 6.23: Relationship between presence of issue and PR acceptance

the issue presence have a 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* do not seem to have an impact on the 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 a lot of similarities. This usually enables use of the same linter for both languages. Moreover, it is not uncommon that projects that are written in C++ also contain some C code and vice versa. The nine selected projects have 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).

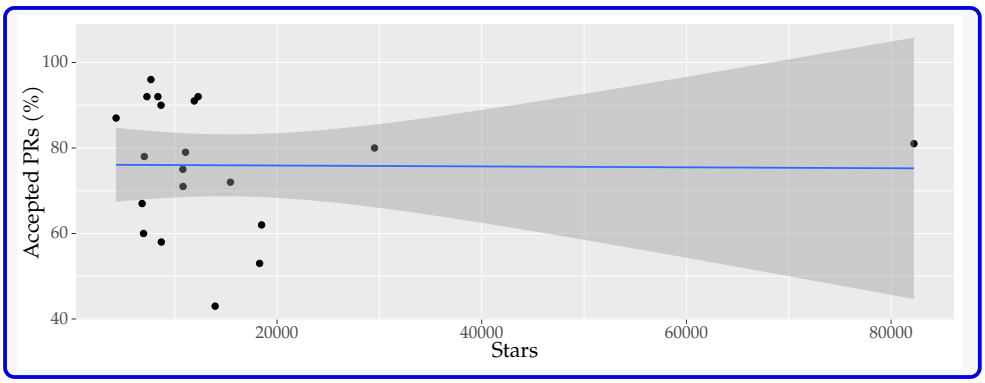

Figure 6.24: Stars and pull request acceptance

I analyzed 8774 C/C++ pull requests. Seventy-seven percent of them have been accepted. The typical pull request introduces 0.25 issues and fixes 0.12 issues; the typical rejected PR introduces 0.79 issues, and the typical accepted PR only 0.15 issues (see Figure 6.25). The 79 % of pull requests did not change the quality of the source code (in terms of the *flawfinder* quality rules).

The most common type of issue is the note (Figure 6.26). The 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 number 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

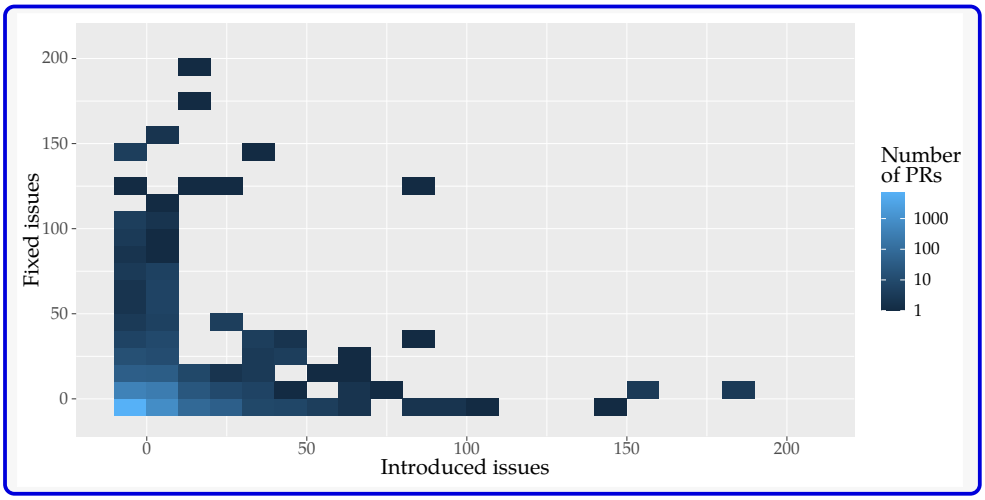

Figure 6.25: Pull requests and quality

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.

As shown in Figure 6.27,classification algorithms rank 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 a large number of false positives). AdaBoost and XGBoost algorithms give this issue importance of 1 %. The average importance is 0.7%. This issue is also most important when only introduced issues are considered. The second most important issue has average importance of only 0.26 %.

The most important error is `buffer-StrCpyNA` ("Does not check for buffer overflows when copying to destination…") with average importance of only 0.9 %. This error is the sixth most important issue. When considering only fixed issues, the `buffer-read` is the most important issue (note); however, the average importance is only 0.28 %.

As can be seen in Figure 6.28, the PCA analysis does not reveal any significant difference between the accepted and rejected pull requests (in terms of code quality).

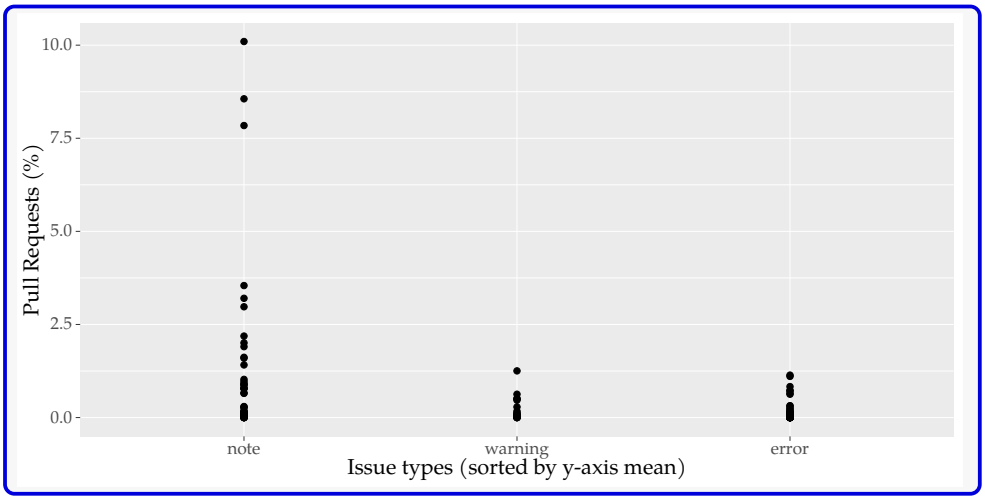

Figure 6.26: flawfinder issues and % of PRs which fixed/introduced them

As shown in Figure 6.29, based on the χ^2 test, the presence of an issue in the PR has a small negative impact on the PR acceptance ($\phi_c = 0.117$). However, When considering only pull requests that solely modified some source files, Cramer's V $\phi_c = 0.024$ and $p = 0.1 > \alpha$ — in this settings, the issue presence does not influence acceptance. Some small impact impact was discovered when the χ^2 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 χ^2 test, the presence of some issue in the PR has a 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 the case of C/C++, the time to close a pull request seems not to be related to found issues. All the models have negative R^2, except the ElasticNet regressor. For the ElasticNet, $MAE = 4681624$ (the 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.

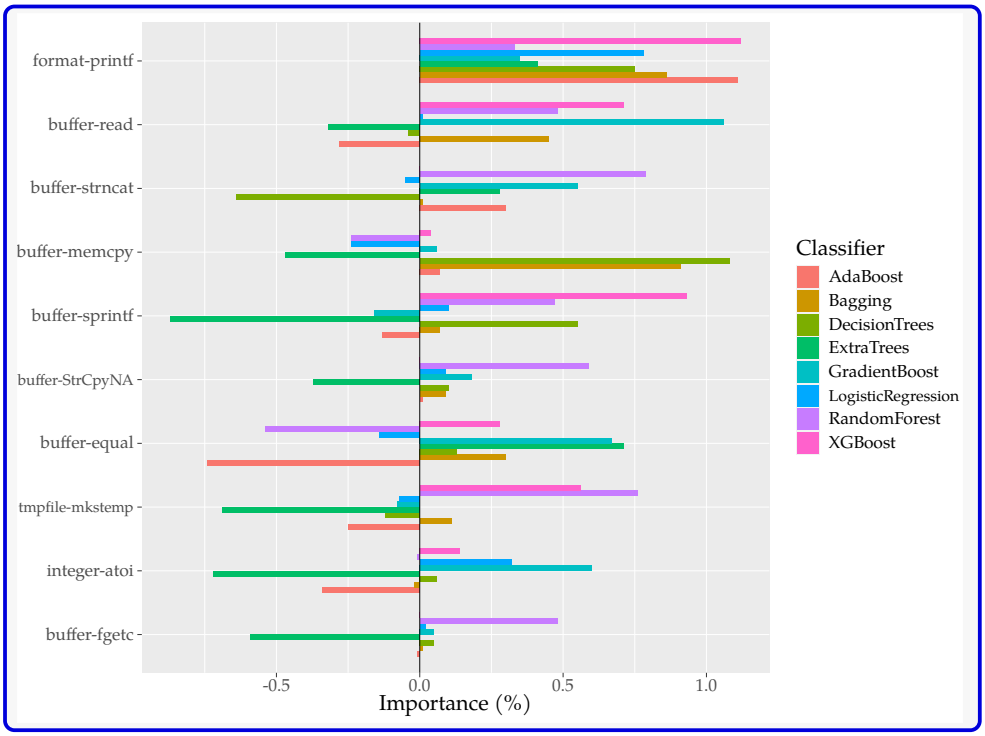

Figure 6.27: Ten most important flawfinder issues

6.6 Programming languages and code quality impact

Comparing the code quality of projects written in different programming languages is a difficult task. Each language has different programming constructs, syntax, and type system. For instance, Python, which is a dynamically-typed multi-paradigm programming language, has completely distinct characteristics from Haskell, which is a purely functional programming language with a strong, static type system.

Moreover, every linter is different and has a 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 a more general-oriented linter that contains a large set of rules for the Java programming language. Lastly, the *HLint* is oriented mainly on code simplification and spotting redundancies.

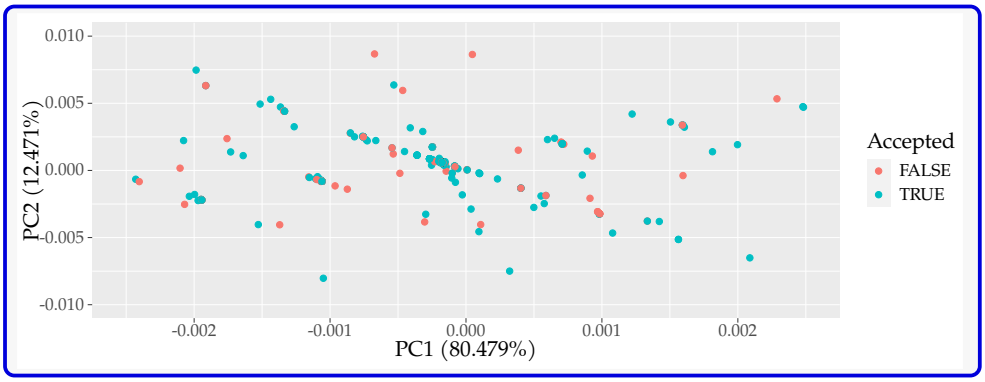

Figure 6.28: PCA scatter plot

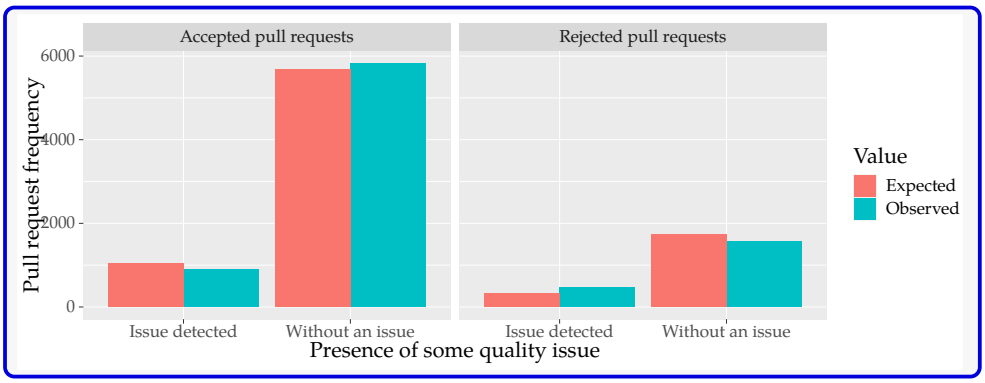

Figure 6.29: Relationship between presence of issue and PR acceptance

On the other hand, there are some metrics that evaluate how effectively trained models predict the acceptance of PR or time to close a PR; and these metrics can be compared across different programming languages. On top of that, the results from the χ^2 test can also be compared. However, the cation is in order because the code quality for each language is evaluated differently, as discussed before. As can be seen in Figure 6.30, in all studied languages, the presence of some issue have a negative effect on the PR acceptance (in terms of χ^2 tests); however, for all of the languages, this effect is small ($\phi_c \approx 0.1$). The smallest effect was observed for Haskell programming language and the highest effect for C/C++. On the other hand, taking into account solely the PRs that only modified some source code of the primary

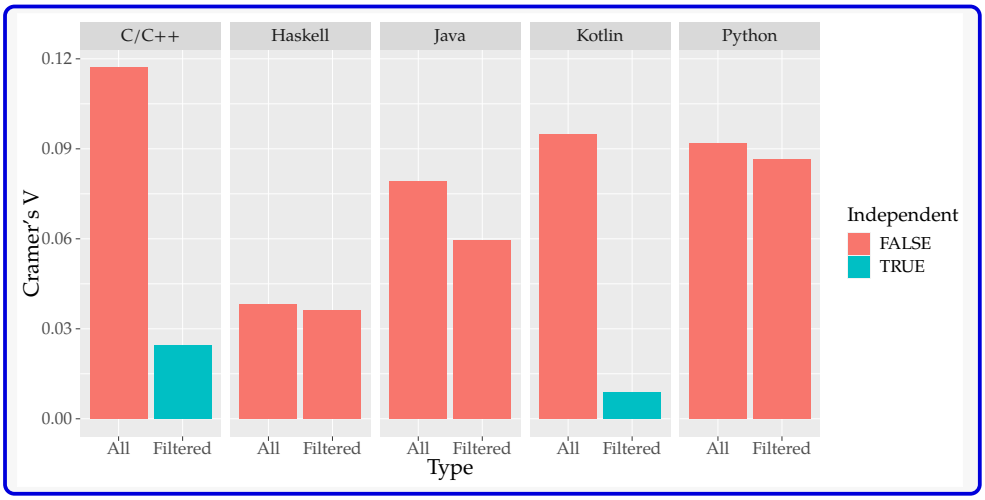

Figure 6.30: Comparison of Cramer's V

language, the χ^2 test indicates that the presence of issue and PR acceptance are independent in the case of the C/C++ and Kotlin. This is a possible threat to validity.

The effect of code quality on acceptance was also studied using classification algorithms. One of the metrics that were used to measure the performance of the classification models is the "area under the ROC curve" (AUC). When using this metric to evaluate models, the Haskell is once again the language when the code quality is least important (Figure 6.31). The average AUC for Haskell models is around 0.5 — the trained models are no better than random guessing. The models for the Kotlin are ranked with the highest AUC score and therefore are better in classification than models for other languages. Except for Haskell, the average AUC is over 0.5 but under 0.6 — these AUC scores are usually considered poor [32]. This indicates that code quality has only a small or no effect on the acceptance.

As can be seen, similar results were obtained for all of the languages. In all of the languages, the code quality impact is small (based on the χ^2 tests and also based on the results from classification algorithms). There is no language that significantly differs from others.

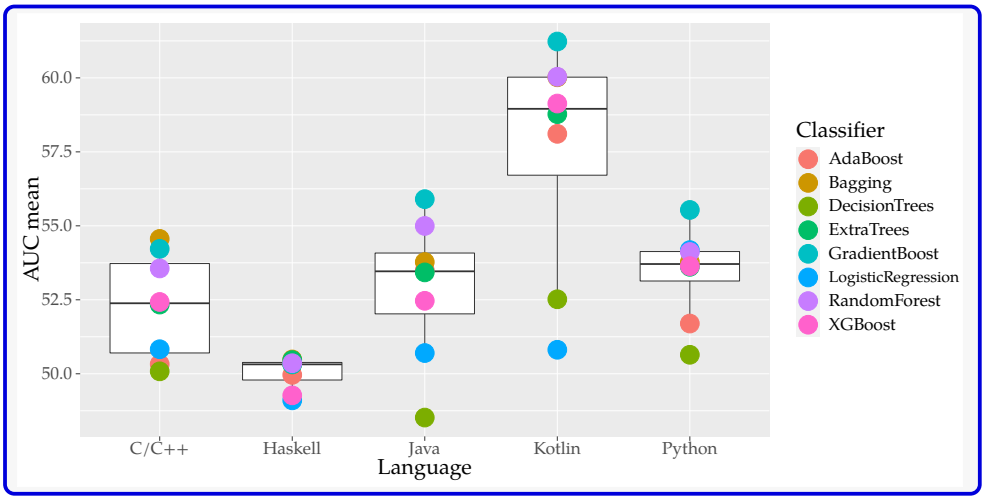

Figure 6.31: AUC for differenct languages (ROC)

As discussed in the previous chapter, there seems to be no connection between the code quality and the time it takes to close pull requests (based on the trained regression models). The smallest MAE was scored by Kotlin models (around 26 days); on the other end of the spectrum are Python models with an average MAE equal to 78.9 days (Figure 6.32). The trained models are unusable, considering that most of the pull requests are closed within the first two weeks (83 % of Kotlin PRs and 76% of Python PRs).

6.7 Threats to validity

The validity of my research is endangered by several things. At first, the selection of the projects is one of the factors that influence the outcomes of the research. This study is focused primarily on popular projects. The rationale behind the project selection is explained in the own dedicated subchapter. It is possible that projects selected using different metrics can yield varying results.

Another possible threat to validity is the selection of pull requests. It is usually not doable to examine all the pull requests of some project. For the projects with a huge number of PRs, the time and computational resources are the limiting factors. Moreover, to examine the

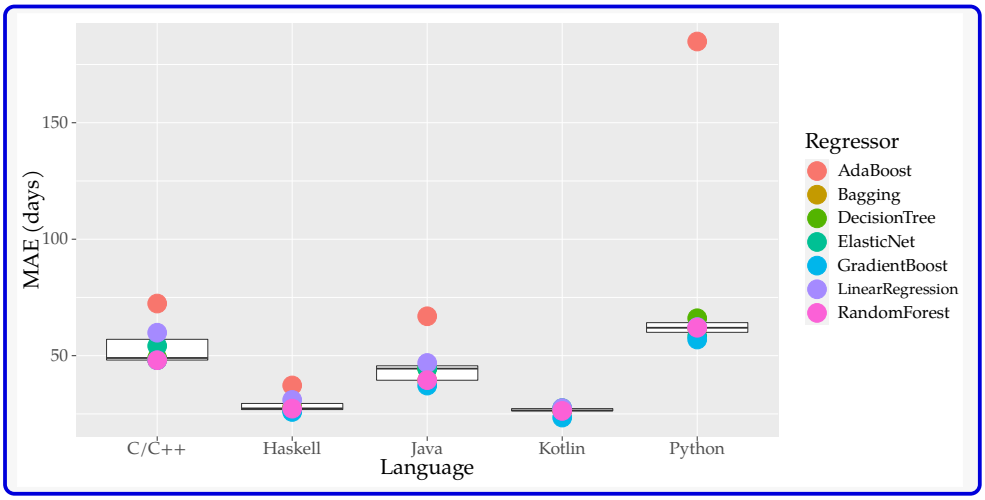

Figure 6.32: Mean absolute error for prediction of time to close a PR

rejected pull requests, the forked repository with the required commits needs to be available. This is not always the case. Sometimes the *force push* can also remove the commits from the accepted pull requests. It is also important to note that linting of some pull requests resulted in an error in the linter, and therefore these PRs were skipped. Pull requests were also skipped if the linting time exceeded the limit (that was set to 1000 seconds) — the PRs that modified a huge number of files were ignored. Furthermore, for some projects, the number of analyzed pull requests was limited to 500 to reduce the total time required for analysis.

Another problem is that pull requests can be merged manually outside GitHub. These pull requests are not recognized as accepted [33]. The projects were selected so that GitHub is the primary way to merge PRs. However, there still can be some PRs merged using alternative methods.

Furthermore, different methods can be used to measure the quality of pull requests. For each programming language, there exist several linters that are focused on a different set of issues, and they can also use different algorithms to detect the same issue. Another possible threat are false positives from linters. The false-positive can arise due to the fact that the files were linted in the isolated environment, and

this can introduce some issues (`import-error`, etc.). Some issues are also hard to detect; for instance, the issue can be specific to some particular context, and the linter does not have to take this context into account. The greatest difficulty with the quality evaluation is the fact that everyone has a unique personal perspective on code quality — code quality does not have a single definition.

The pull requests sometimes contain also files that are not written in the primary programming language of the project. The pull request then can be rejected because of these files.

Lastly, there are several factors that influence PR acceptance. Some of them were discussed in previous chapters (number of commits, submitter's status, etc.). The one factor that influences the acceptance is a number of lines that were changed [10]. The more lines are added/changed, the higher the probability that the pull request will be rejected, but the chance that some quality issue will be introduced is also higher. In this case, it is difficult to distinguish if the pull request was rejected because of the code quality or because the changes are too big.

7 Conclusion

I analyzed 41576 pull requests from 100 projects written in 5 different programming languages (Python, Java, Kotlin, Haskell, C/C++) to study the relationship between the code quality and pull request acceptance. The quality of the individual pull requests was measured using static code analysis.

Almost half of the analyzed PRs introduced some code quality issue, and 31 % of pull requests fixed some issue. However, data differs significantly between the languages (because different static analysis tools were used). In C/C++ projects, only 16 % of pull requests introduced some issue, while in Java, almost 76 % of PRs. The proportion of accepted pull requests was different for each project; however, on average, 76 % of PRs were accepted.

Several statistic techniques were used to understand if the code quality affects PR acceptance. For each language, the χ^2 test of independence was performed, and the number of accepted PRs without a code quality issue was always higher than expected. However, in all languages, the impact on acceptance was only small ($\phi_c \approx 0.1$).

Multiple classification algorithms were used to predict the pull request acceptance using the code quality. However, all of them performed poorly ($AUROC < 0.6$). The most problematic was the Haskell language — all models were no better than a random predictor ($AUROC \approx 0.5$).

The trained models were also used to understand the importance of individual issues. Unfortunately, no issue with a significant impact on the PR acceptance was detected. All discovered issues have average importance below 1 % (between all models).

The influence of code quality on time to close a PR was also studied. Several regression models were trained to predict this time. However, all of the trained models have very high *mean absolute error*: around one month. This makes models worthless because most of the PRs are closed within two weeks.

To conclude, the poor code quality seems to have a small negative impact on the pull request acceptance. However, there seems to be no effect on the time it takes to close a PR.

7.1 Comparison with related work

Best of my knowledge, there is only one study [2] about the effect of quality flaws on the pull request acceptance. Lenarduzzi et al. analyzed 28 well-known Java projects. I reused the script they provided for PR classification and also applied similar statistical techniques so that my findings could be compared with theirs. The χ^2 test of independence yielded similar results (they obtained $\phi_c = 0.12$). My classification models have slightly better performance (mean *AUROC* is higher by 0.023). The difference in performance can be caused by various factors — project selection and the ratio of accepted pull requests (only 53 % of PRs they studied were accepted). The code quality was evaluated using the same linter (PMD). However, I also took into account issues that were fixed by the PR. Moreover, a different technique was used to identify issues that were introduced in the PR (they used diff-files provided by GitHub API). Similar to my findings, Lenarduzzi et al. did not identify any particular issues that have a significant effect on the acceptance.

I extended the work of Lenarduzzi with an analysis of four new programming languages (Python, Kotlin, Haskell, and C/C++). I also added the analysis of the delivery time of pull requests, and as far as I know, this is the first study that researches the relationship between the code quality and the time it takes to close a pull request.

7.2 Future work

I consider my work complete. However, there is still plenty of possibilities for how to improve and expand my work. Several improvements can be made to obtain more reliable data for analysis. If the pull request is not merged using GitHub, then the PR is incorrectly classified as rejected. It is possible to utilize some heuristics that will recognize merging through plain Git utilities.

For the proper quality evaluation, the linter choice is essential. Each linter is focused on some specific set of issues, and this can introduce some form of bias. It would be beneficial to use multiple linters for one programming language. The linters used for C/C++ and Kotlin are not very sophisticated. However, adding a more advanced lin-

ter for C/C++ is complicated — the state-of-the-art linters require information about compiler flags. This information cannot always be retrieved automatically (from makefiles). Therefore, a lot of pull requests require manual customization.

Some projects use linters as part of the *continuous integration* or during the build process. Additional research needs to be performed to understand if the maintainers of those projects are more strict about the code quality, and therefore the effect on the PR acceptance is larger.

Appendix

This appendix contains additional plots and tables. It also discusses the scripts that were used for statistical analysis of pull requests.

Scripts used for analysis

In order to simplify the analysis of retrieved data, I created the script (`pr_process.py`) 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 contain any source code written in the primary language or PRs that contain corrupted files (the linter was unable to analyze those files).

The converted data about the pull request were subsequently analyzed in order to answer my research questions. For the classification (RQ_2) was used the Python script[1] (`pr_classification.py`) provided by Lenurdazzi et al. [2].

I also created the script (`pr_reqression.py`) that runs the regression algorithms on the data in order to answer RQ_4. This script is written in Python, and it uses scikit-learn[2] library.

The rest of the analysis was done using the `analysis.R`. This small R program imports the data generated by other scripts. These data are then analyzed using various statistical methods. The script is also used to plot graphs, create tables and then export them directly into the LaTeX.

Additional plots and tables

This chapter contains tables that summarize the information about used projects and issue categories. Furthermore, the chapter visualizes metrics that were used to evaluate classification and regression algorithms.

1. `https://figshare.com/s/d47b6f238b5c92430dd7`
2. `https://scikit-learn.org/stable/index.html`

Projects summary

The following tables contain information about analyzed projects. Columns named *Introduced/Fixed issues* indicate the average number of issues per pull request, and the values were computed using a 5% trimmed mean.

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

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

Issue categories

The Following tables show the issue types and the total number of issues that were fixed/introduced and belong to the given category. Furthermore, tables show how many pull requests fixed/introduced issues of the given category.

Table 7.6: Pylint issue categories

Category	Introduced in total	Introduced by PR	Fixed in total	Fixed by PR
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	Introduced by PR	Fixed in total	Fixed by PR
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

Table 7.8: HLint issue categories

Category	Introduced in total	Introduced by PR	Fixed in total	Fixed by PR
Warning	4156	1143	3262	787
Suggestion	7874	1500	8212	986
Error	27	14	33	14

Table 7.9: flawfinder issue categories

Category	Introduced in total	Introduced by PR	Fixed in total	Fixed by PR
note	15874	1275	16807	901
error	1861	297	1406	206
warning	487	151	541	146

Classification metrics

Following metrics were used to evaluate classification algorithms.

Table 7.10: Python classification metrics

Classifier	AUC	Precision	Recall	MCC	F-Measure
AdaBoost	0.517	0.7381	0.9375	0.069	0.8254
Bagging	0.5378	0.7444	0.8924	0.0817	0.8113
DecisionTrees	0.5064	0.7453	0.8224	0.0611	0.7818
ExtraTrees	0.5361	0.74	0.8936	0.0582	0.8092
GradientBoost	0.5554	0.7381	0.9617	0.0755	0.835
LogisticRegression	0.5418	0.7386	0.9798	0.0798	0.8423
RandomForest	0.5411	0.741	0.9189	0.0735	0.8199
XGBoost	0.5364	0.7411	0.9252	0.0773	0.8223

Table 7.11: Java classification metrics

Classifier	AUC	Precision	Recall	MCC	F-Measure
AdaBoost	0.535	0.744	0.9224	0.1184	0.8235
Bagging	0.5378	0.7331	0.8366	0.0394	0.7802
DecisionTrees	0.4852	0.7315	0.7314	0.0231	0.7311
ExtraTrees	0.5343	0.7379	0.8679	0.0689	0.7972
GradientBoost	0.559	0.7423	0.9543	0.1311	0.8349
LogisticRegression	0.507	0.7361	0.9655	0.0846	0.8352
RandomForest	0.55	0.7362	0.8879	0.073	0.8046
XGBoost	0.5247	0.736	0.8752	0.0761	0.799

Table 7.12: Kotlin classification metrics

Classifier	AUC	Precision	Recall	MCC	F-Measure
AdaBoost	0.5811	0.8207	0.9695	0.1819	0.8888
Bagging	0.6002	0.8216	0.9278	0.1414	0.8714
DecisionTrees	0.5252	0.821	0.8762	0.1112	0.8475
ExtraTrees	0.5877	0.8215	0.9316	0.1369	0.873
GradientBoost	0.6123	0.8177	0.9758	0.1446	0.8896
LogisticRegression	0.5081	0.8069	0.9959	0.0754	0.8914
RandomForest	0.6004	0.8194	0.944	0.1333	0.8772
XGBoost	0.5913	0.8175	0.9646	0.1427	0.8848

Table 7.13: Haskell classification metrics

Classifier	AUC	Precision	Recall	MCC	F-Measure
AdaBoost	0.4996	0.7771	0.9764	0.0241	0.8654
Bagging	0.5048	0.7777	0.9551	0.0202	0.8572
DecisionTrees	0.5032	0.7772	0.9271	0.0119	0.8452
ExtraTrees	0.5046	0.7779	0.9494	0.0237	0.855
GradientBoost	0.5032	0.7775	0.9907	0.0283	0.8712
LogisticRegression	0.491	0.7759	0.9902	0.004	0.87
RandomForest	0.5036	0.777	0.9616	0.0168	0.8594
XGBoost	0.4927	0.778	0.9822	0.0399	0.8682

Table 7.14: C/C++ classification metrics

Classifier	AUC	Precision	Recall	MCC	F-Measure
AdaBoost	0.5033	0.7711	0.9792	0.0707	0.8627
Bagging	0.5456	0.7774	0.9679	0.1159	0.8622
DecisionTrees	0.5008	0.7764	0.9527	0.0965	0.8555
ExtraTrees	0.5234	0.7742	0.9637	0.0865	0.8585
GradientBoost	0.5422	0.773	0.9884	0.1117	0.8675
LogisticRegression	0.5083	0.769	0.9918	0.0586	0.8663
RandomForest	0.5356	0.7756	0.9747	0.1131	0.8638
XGBoost	0.5243	0.774	0.9808	0.1052	0.8652

ROC curves

For each classifier, the *receiver operating characteristic curve* (ROC) was plotted. The average values over folds were used for the ROC.

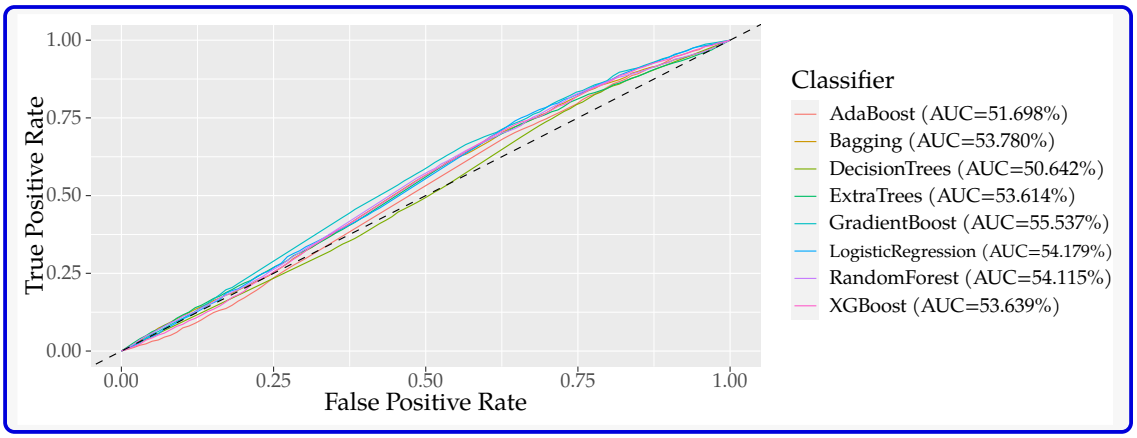

Figure 7.1: Python classification ROC

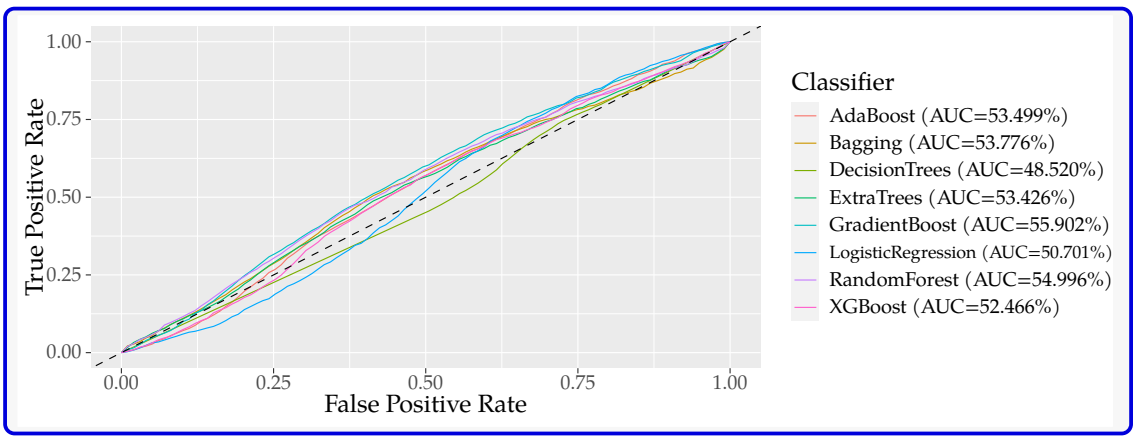

Figure 7.2: Java classification ROC

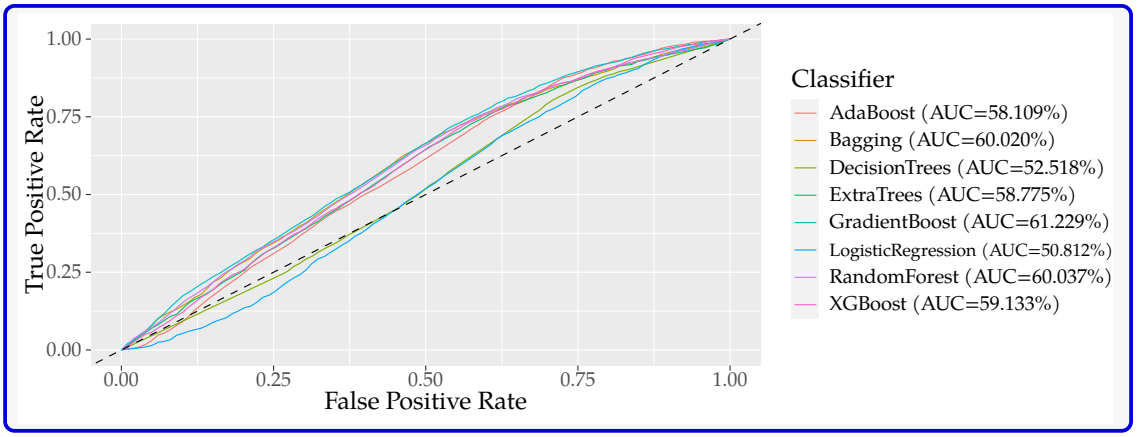

Figure 7.3: Kotlin classification ROC

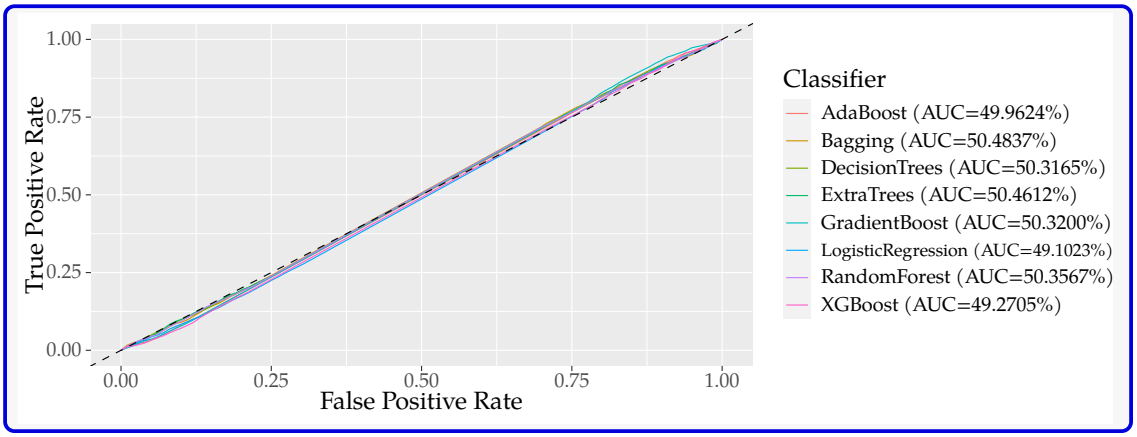

Figure 7.4: Haskell classification ROC

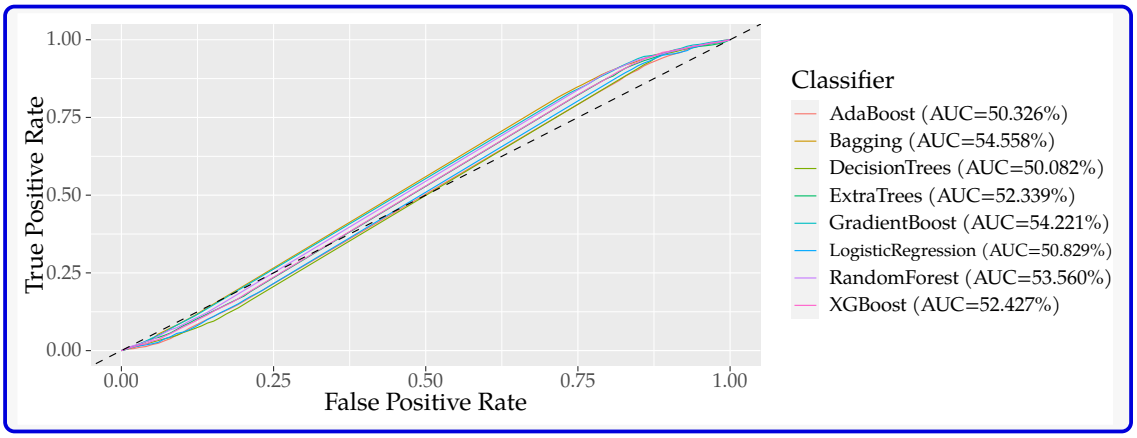

Figure 7.5: C/C++ classification ROC

Regression metrics

Following metrics were used to evaluate regression algorithms. EV stands for *explained variance* — this metric is similar to R^2, but subtracts mean error from the sum of squared residuals.

Table 7.15: Python regression metrics

Regressor	MAE	MSE	R^2	EV
AdaBoost	15973460	329401637686959	-1.5507	-0.2215
Bagging	5336084	157847854956052	-0.2223	-0.2208
DecisionTree	5708585	223751143542147	-0.7326	-0.7317
ElasticNet	5036178	129277016827251	-0.0011	-0.0011
GradientBoost	4918887	129533128855813	-0.003	-0.003
LinearRegression	5381938	139399638460786	-0.0794	-0.0794
RandomForest	5355559	158248066602028	-0.2254	-0.2238

Table 7.16: Java regression metrics

Regressor	MAE	MSE	R^2	EV
AdaBoost	5781968	121359910802985	0.3639	0.4028
Bagging	3405617	110721713131779	0.4196	0.4196
DecisionTree	3840642	172877663509287	0.0938	0.0939
ElasticNet	3840132	173681710969993	0.0896	0.0896
GradientBoost	3210541	109039175913116	0.4285	0.4287
LinearRegression	4049121	166792691079607	0.1257	0.1258
RandomForest	3412341	110834998352984	0.419	0.4191

Table 7.17: Kotlin regression metrics

Regressor	MAE	MSE	R^2	EV
AdaBoost	2338813	33550793797470	0.1295	0.1318
Bagging	2277309	27920475003210	0.2756	0.2817
DecisionTree	2288615	40851993169579	-0.0599	-0.0577
ElasticNet	2375866	32413611388373	0.159	0.1596
GradientBoost	2024497	23668194119907	0.3859	0.3874
LinearRegression	2375121	32221018909680	0.164	0.1647
RandomForest	2271417	27681011966890	0.2818	0.2876

Table 7.18: Haskell regression metrics

Regressor	MAE	MSE	R^2	EV
AdaBoost	3215222	56655016796029	-0.0208	-1e-04
Bagging	2363540	58365649395896	-0.0516	-0.0514
DecisionTree	2416772	69338005300816	-0.2493	-0.2488
ElasticNet	2286860	55883566536222	-0.0069	-0.0063
GradientBoost	2233096	66379435207705	-0.196	-0.1958
LinearRegression	2678970	72626066459140	-0.3085	-0.3077
RandomForest	2355006	58464125540702	-0.0533	-0.0532

Table 7.19: C/C++ regression metrics

Regressor	MAE	MSE	R^2	EV
AdaBoost	6253848	101173526849545	-0.1147	-0.0171
Bagging	4162859	92614187333881	-0.0204	-0.017
DecisionTree	4234453	113821369753924	-0.254	-0.2511
ElasticNet	4681624	89914644202604	0.0094	0.0103
GradientBoost	4153060	93185095307485	-0.0266	-0.0256
LinearRegression	5172189	190936368069504	-1.1036	-1.1017
RandomForest	4152398	92335449872508	-0.0173	-0.014

Absolute error density

The regression models were evaluated on a testing data set. For every predicted value, the absolute error was computed. The following figures show the density curves for absolute errors.

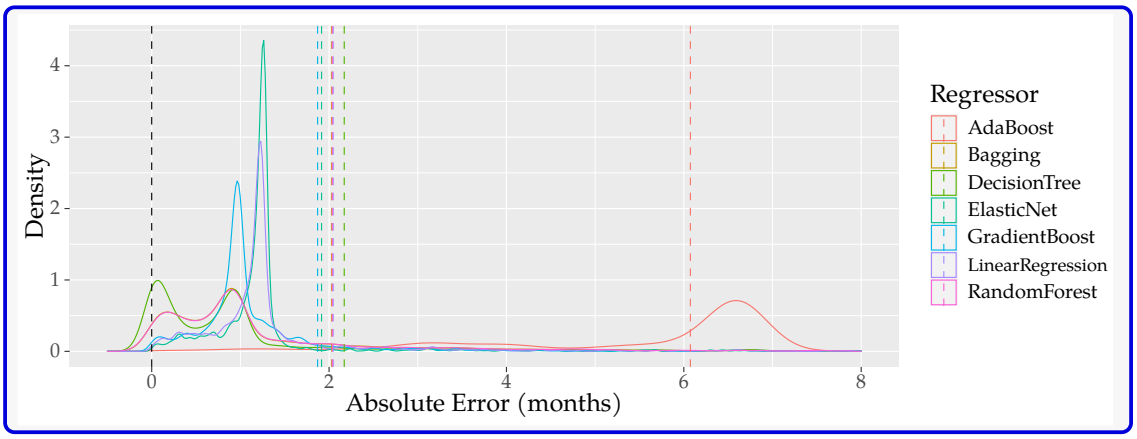

Figure 7.6: Python AE density

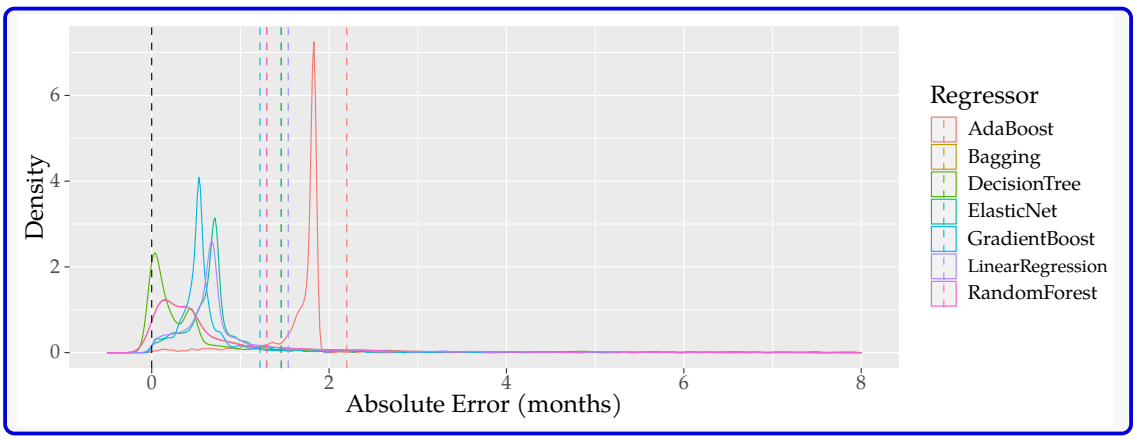

Figure 7.7: Java AE density

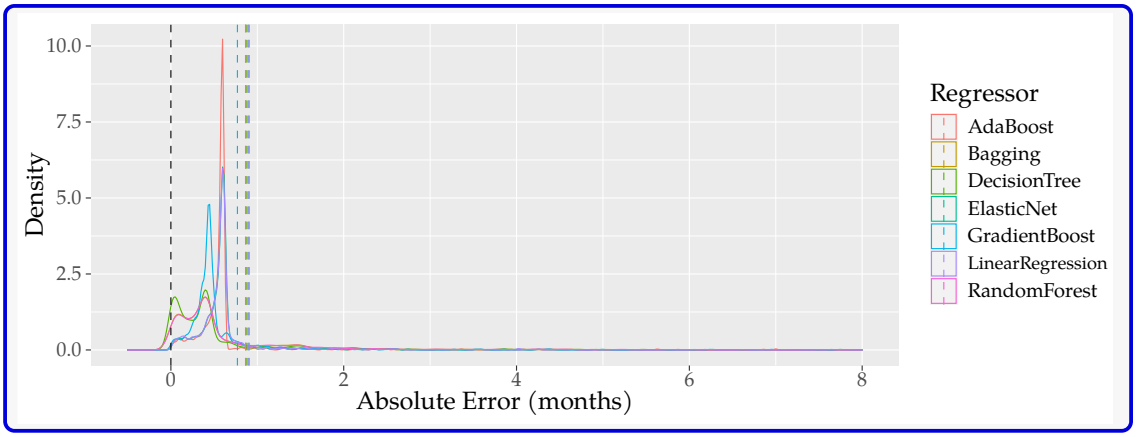

Figure 7.8: Kotlin AE density

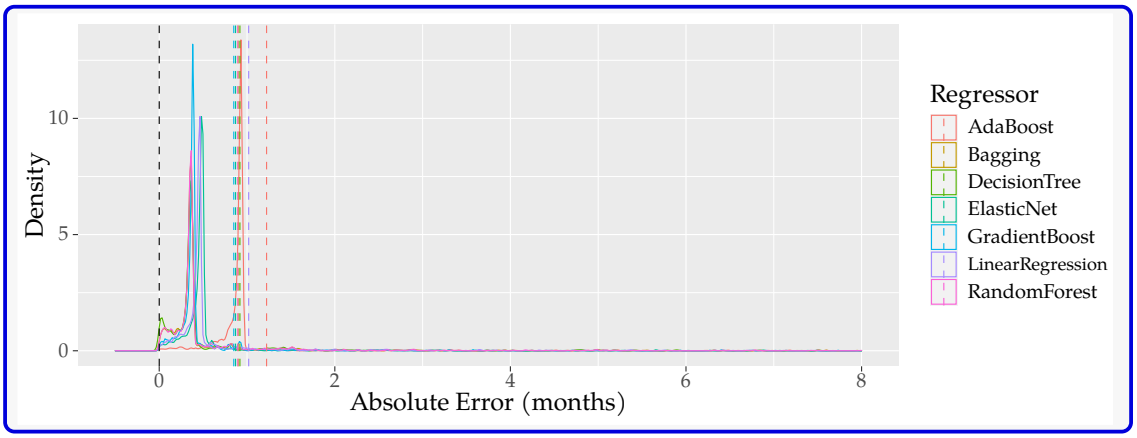

Figure 7.9: Haskell AE density

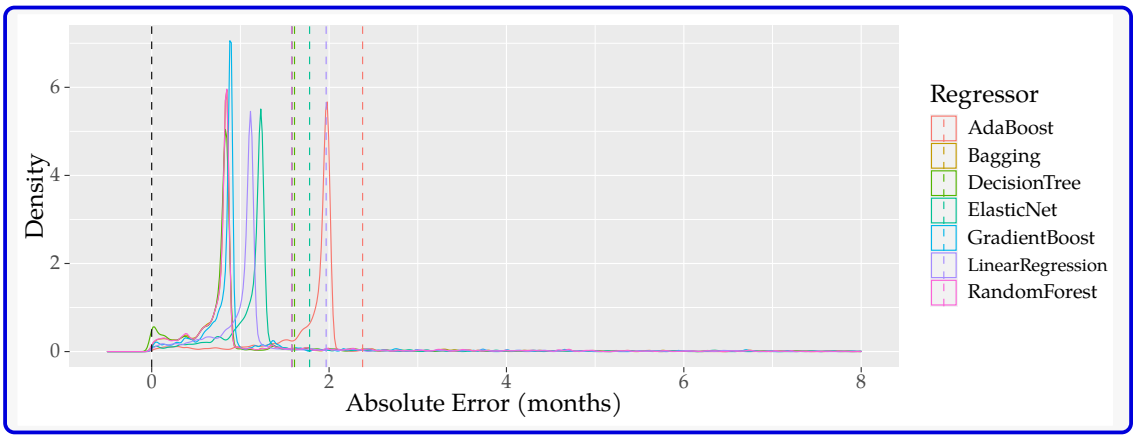

Figure 7.10: C/C++ AE density

Bibliography

1. 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.
2. 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.
3. 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.
4. GRAHAM, D.; VAN VEENENDAAL, E. *Foundations of software testing: ISTQB certification*. 2nd ed. International Thomson Business Press, 2008. ISBN 1844809897.
5. ISO/IEC 25010:2011. *Systems and software engineering — Systems and software Quality Requirements and Evaluation — System and software quality models*. 2011. Standard. International Organization for Standardization.
6. 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.
7. 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.

8. 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.

9. ALAMI, A.; COHN, L.; WĄISOWSKI, 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.

10. 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.

11. 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.

12. 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.

13. 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.

14. 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.

15. 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.

16. 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.

17. 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.

18. 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.

19. 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.

20. GOUSIOS, G. The GHTorrent dataset and tool suite. In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE, 2013, pp. 233–236. Available also from: https://dl.acm.org/doi/10.5555/2487085.2487132.

21. KLEINBAUM, D.; DIETZ, K.; GAIL, M.; KLEIN, M. *Logistic regression*. Springer, 2002. Available from DOI: 10.1007/978-1-4419-1742-3.

22. KINGSFORD, C.; SALZBERG, S. What are decision trees? *Nature biotechnology*. 2008, vol. 26, no. 9, pp. 1011–1013. Available from DOI: 10.1038/nbt0908-1011.

23. BREIMAN, L. Bagging predictors. *Machine learning*. 1996, vol. 24, no. 2, pp. 123–140. Available from DOI: 10.1007/BF00058655.

24. BREIMAN, L. Random forests. *Machine learning*. 2001, vol. 45, no. 1, pp. 5–32. Available from DOI: 10.1023/A:1010933404324.

25. GEURTS, P.; ERNST, D.; WEHENKEL, L. Extremely randomized trees. *Machine learning*. 2006, vol. 63, no. 1, pp. 3–42. Available from DOI: 10.1007/s10994-006-6226-1.

26. SCHAPIRE, R. Explaining adaboost. In: *Empirical inference*. Springer, 2013, pp. 37–52. Available from DOI: 10.1007/978-3-642-41136-6_5.

27. FRIEDMAN, J. Stochastic gradient boosting. *Computational Statistics Data Analysis*. 2002, vol. 38, no. 4, pp. 367–378. Available from DOI: 10.1016/S0167-9473(01)00065-2.

28. CHEN, T.; GUESTRIN, C. XGBoost: A Scalable Tree Boosting System. In: *Proceedings of the 22nd International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 785–794. Available from DOI: 10.1145/2939672.2939785.

29. COCHRAN, W. The χ^2 Test of Goodness of Fit. *The Annals of Mathematical Statistics*. 1952, vol. 23, pp. 315–345. Available from DOI: 10.1214/aoms/1177729380.

30. WEISBERG, S. *Applied linear regression*. Wiley, 2005. Wiley Series in Probability and Statistics. Available from DOI: 10.1002/0471704091.

31. ZOU, H.; HASTIE, T. Regularization and variable selection via the elastic net. *Journal of the royal statistical society*. 2005, vol. 67, no. 2, pp. 301–320. Available from DOI: 10.1111/j.1467-9868.2005.00503.x.

32. LEMESHOW, S.; STURDIVANT, R.; HOSMER, D. *Applied Logistic Regression*. Wiley, 2013. Wiley Series in Probability and Statistics. Available from DOI: 10.1002/9781118548387.

33. KALLIAMVAKOU, E.; GOUSIOS, G.; BLINCOE, K.; SINGER, L.; GERMAN, D.; DAMIAN, D. The Promises and Perils of Mining GitHub. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 92–101. Available from DOI: 10.1145/2597073.2597074.