University of Zurich[UZH]

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Toward a Definition of Fairness to Calculate Contributors Performance in Git Repositories

Norina Braun
Zurich, Switzerland
Student ID: 18-706-531

Supervisor: Dr. Bruno Rodrigues, Dr. Eder Scheid,
Prof. Dr. Burkhard Stiller
Date of Submission: 07.09.2022

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmuehlestrasse 14, CH-8050 Zurich, Switzerland
URL: http://www.csg.uzh.ch/

Zusammenfassung

In der heutigen Zeit finden sich in praktisch jedem Quellcode Anteile von Open Source Software. Allerdings beinhalten die Mehrheit der Quellcodes auch veraltete Komponenten, was zu Problemen führen kann. Um diese zu lindern, hat man begonnen Open Source Projekte finanziell zu unterstützen, um so die Entwickler dazu zu bringen sich längerfristig an einem Projekt zu beteiligen und für dessen Unterhalt zu sorgen. Die Mehrheit solcher Sponsoringplattformen konzentriert sich jedoch darauf Spenden zu den Projekten zu bringen und überlässt die Entscheidung wie diese unter den Entwicklern aufgeteilt werden sollte den Projektleitern. Diese Arbeit stellt eine Methode vor um den Beitrag eines Entwicklers zu einem Open Source GitHub Repository automatisch zu berechnen. Es wird eine Contribution Engine entwickelt, die die Anzahl Codezeilen als Grundlage nimmt und dann aufgrund von Bewertungskennzahlen in den Bereichen Instandhaltbarkeit, Fehlerbereinigung und Testabdeckung einen Bonus aufaddiert. Ausserdem lässt sich die Berechnung der Beteiligung indivdualisieren durch eine anpassbare Gewichtung der Metriken. Die Engine wurde mit Hilfe eines Universtätsprojektes entwickelt und mit einem ähnlichen Projekt verglichen um die Zulässigkeit zu zeigen. Abschliessend wurde gezeigt, dass die Contribution Engine die beiden betrachteten Projekte erfolgreich analysieren konnte, dass jedoch die Bewertungskriterien zu spezifisch an die Projekte angepasst waren um sie verallgemeinern zu können. Ausserdem stellte sich heraus, dass mit der hier betrachteten Definition von Fairness keine komplette Automatisierung des Evaluierungsprozesses möglich ist.

Abstract

In today's world, open-source software components are present in almost any codebase. However, the majority of these codebases also contain outdated components, which can lead to problems. To mitigate this, there has been a shift toward financially funding open-source projects to increase developers' commitment to maintaining the projects. However, the majority of sponsorship platforms focus on bringing funds to the projects and leave the decision of how to distribute it among the developers to the project owners. This thesis provides a method for automatically estimating a developer's contribution to an open-source GitHub repository. A contribution engine is developed that takes lines of code as a baseline for the contribution and considers the metrics maintainability, bug fixes, and test coverage to add a bonus and allows for customizability by adapting the importance of the metrics. The engine is implemented with the help of a university project and tested against a similar project to demonstrate its legitimacy. It was concluded that the proposed evaluation engine succeeds in fairly evaluating code contribution in the considered projects, but the metrics are highly subjective and thus cannot be generalized to different projects. Furthermore it was found that entirely automating the evaluation process was not possible within the considered definition of fairness.

Acknowledgments

First and foremost I would like to thank Dr. Bruno Rodrigues for bringing the topic to my attention and then continuously supporting and motivating me through every aspect of this project. I would also like to thank Prof. Dr. Burkhard Stiller for giving me the opportunity to write this thesis as part of the Communication Systems Group of the University of Zurich. Finally I would like to thank Dr. Eder Scheid for supervising the project.

Contents

Chapter 1

Introduction

Open-source Software (OSS) has become ubiquitous in the modern world. 97% of all codebases nowadays at least partly rely on an open-source component, for some projects open-source accounts for the majority of the code [20]. Traditionally these OSS parts have been developed by unpaid volunteers, that mostly work alone or in small teams. According to a 2022 census by Harvard [29], in 94% of all open-source projects a small team of fewer than 10 people is responsible for 90% of the code, and in 23% of projects a single creator is responsible for 80% of the code. Such projects have a low truck factor and are dependant on the personal lives of the contributors. If the developers loose motivation to maintain the projects or don't have time to work on them anymore, the components are abandoned and deprecate over time, which can lead to problems for other projects, that utilize these abandoned parts in their own codebases. This has become increasingly problematic with the widespread usage of OSS, and according to the 2022 synopsys study [20] 88% of the considered code-bases included outdated open-source components. This makes the components vulnerable and can lead to security risks, hence why it is important to motivate the developers to maintain their codebases.

As mentioned above, the developers that worked on open source projects were usually not compensated at all for their contributions and mostly did it because they were personally invested in the projects or wanted to improve their Curriculum Vitae (CV). In this regard, intrinsic and altruistic motivations have been for many years the driving motivation for OSS contributors, but the increased complexity that often requires more time for relevant contributions has also exposed a need for extrinsic (mainly financial) incentives. In recent years there has been a shift so that people can get paid for the hard work they put in. Since they can get reimbursed for the time and effort spent on such projects, developers are able to devote more time to projects they are enthusiastic about and can earn part of their income by working on open-source projects. Since maintaining and refining a codebase now can get monetarily rewarded, committing to a project for a longer time period can be a viable source of income, and developers should be less likely to abandon projects due to time constraints.

To establish this paradigm on a grand scale, platforms and services are needed, that supply the infrastructure for the donations. There are already several platforms and programs that have been created to provide a possibility to sponsor projects or developers directly.

In early 2019 the Linux foundation introduced their platform CommunityBridge [8] and just a few months later Github announced their sponsoring program GitHub Sponsors [9]. Both of these programs allow donations to developers or projects. Another platform purely designed to provide a sustainable income for contributors working on open-source project is the FlatFeeStack [31] project. However, the possibility to raise money and gain supporters solves only half of the problem. Another key challenge is then deciding how the money should be distributed among the developers. When sponsoring creators directly, this is not a problem, but with the approach to support a project rather than a person, it becomes relevant. The simplest solution would be to just let the project owner decide how the money should be distributed. However, not only does this cost time and effort from the project leader, but it also means that the decision will likely be biased. A more elegant solution would be an automatic evaluation engine, which periodically assesses a repository and calculates the cut of a donation for each contributor, which is proportional to the amount of work they contributed to the corresponding project.

Thus designing such a contribution engine that determines the percentage for each developer is a key challenge to fairly distributing the donation and building up trust in the platform. If a user wants to work part or full time on open-source, they need a reliable system that provides a sustainable income. They need to trust that the contribution engine evaluates their work fairly and that their time and effort are appreciated. Hence building a trustworthy contribution engine is of great importance for a sponsoring platform to be viable. The development and evaluation of a fair contribution engine will be the focus of this thesis.

1.1 Thesis Goals

During the course of the thesis the following bachelor thesis questions are required to be answered:

- How to extract metrics or indicators of contributors' performance from an open-source Git repository?
- How to define a fairness concept to calculate the importance of contributions in Git repositories based on metrics and indicators?

1.2 Methodology

It is a known problem that most open-source projects contain components that are outdated [20]. As an approach to mitigate this, there have been advances in building platforms to sponsor projects and monetarily reward the developers to entice them to maintain the projects. To calculate the percentage of a donation per developer, several approaches 2 have been introduced. Based on the metrics and findings of other studies, in this thesis a contribution engine was developed and used to analyze two university projects.

1.3 Thesis Outline

Chapter 2 introduces background knowledge about **Fairness** and the **Metrics** that indicate a contribution towards the project goal. Chapter 3 describes the requirements to meet the goals and shows the architecture behind the different components that should fulfill those. It furthermore includes information about the implementation of the **Contribution Engine**. Chapter 4 compares two university projects from the SoPra course with each other, evaluates them according to different metrics and discusses the impact of said metrics. Finally Chapter 5 concludes the thesis in a final consideration including a summary, conclusions and future work.

Chapter 2

Fundamentals

2.1 Background

In order to build a tool to fairly assess a contribution to a Git repository, we first need to establish a concept of how this contribution should be evaluated. To introduce a notion of fairness, we need to consider not only the amount of code a developer contributes (*i.e.*, quantitative metrics), but also the quality (*i.e.,* qualitative metrics) and the importance of it (*i.e.*, a weight for each metric).

2.1.1 Fairness Theory

This thesis aims to develop a contribution engine that fairly assesses a developers contribution to a Git repository. Therefore, we need to establish a concept of fairness in this context. If we use the Cambridge dictionary definition of fairness [11], it means treating people equally. Since the evaluation is done with a mathematical function, which is defined for a given project, everybody working on said project will be evaluated by the same standard and thus fairness is inherently given.

However, even if a system is fair in theory, that doesn't always mean it will be perceived as fair by the users. As this study [25] suggests, there is a correlation between the perceived fairness and the accuracy of a performance evaluation. Manually reviewing code also comes with some perceived biases, as according to [14] almost 40% of the participants reported to have been evaluated unfairly and just as many admit to having evaluated someone else's code unfairly.

Perceived fairness has also been studied in the context of participation grading in a university classroom. This study [30] found that the three factors which influence perceived fairness the most are: explicitness of grading criteria, frequency of feedback, and proactiveness of instructor techniques. While this study may not necessarily be directly applicable in the context of code evaluation, it showcases the need for a transparent evaluation scheme. This way the person to be evaluated understands the criteria and gets some feedback, which allows them to improve. This focus on clear feedback is also often given as

a tip to improve employee evaluation, eg. [17] or [5] While these studies mostly relate to evaluations performed by humans and therefore are subject to human biases, they highlight the importance of building an evaluation system the developers trust. The challenge hereby is to find a balance between giving the developer adequate feedback and the chance to improve and allowing them to trick the system and submit their work in such a way that it benefits their evaluation score, while not benefitting the system they work at.

As we have seen, fairness is not a quantitative measure, but it is highly subjective. Thus it is hard to define fairness metrics to evaluate the judgment of an contribution engine. Additionally different projects value different metrics more highly. A project in a classroom environment will likely be evaluated differently than a project in the open market. To allow for this flexibility, the evaluation engine of this thesis, will provide the possibility to include and exclude certain metrics and to adjust the weights for every metrics, such that the importance of any metric can be personalized for every project.

2.1.2 Git

Git [15] is the version control system upon which GitHub [16] is built. This thesis will be limited to open source projects with a public GitHub repository, since they provide the necessary data for the evaluation engine. As GitHub is a popular platform for open source projects hosting millions of projects [16], it is a reasonable decision to focus on GitHub repositories.

2.1.3 Metrics

1. **Lines of Code**
   The contribution engine measures the amount every developer has contributed to a repository. The baseline for such a contribution is the code they commit, and thus the primary metric is the amount of code they write, i.e. lines of code (LOC). In a next step the quality of their contribution is evaluated by considering some quality metrics.

2. **Code quality**
   Code Quality is a broad term that cannot be easily defined and assigned a numerical score. However, the international organization for standardization (ISO) and the international electrotechnical commission (IEC) have developed a standard to describe system and software quality [21] The standard is composed of two different models, a quality in use model, as well as a product quality model. The quality in use model considers five characteristics: effectiveness, efficiency, satisfaction, freedom from risk and context coverage. It is mainly concerned with the impact of the product on the people interacting with it. The product quality model on the other hand evaluates eight characteristics: functional stability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability. It is more focused on the code and less on the human-computer interaction. Both models are further split into sub-characteristics, that break down the categories and explore more detailed

aspects. But since the first model depends on user experience, it is not of much use for this thesis. The product quality model, however, is a useful basis for showing the correctness of our quality metrics. Its sub-categories and whether or not they are applicable for this thesis are discussed now:

- **Functional suitability** includes *Functional completeness, Functional correctness* and *Functional appropriateness*. Functional completeness could be measured with the GitHub issue tracking system, if a given projects uses issues. Whenever an issue is closed, it means that the specified tasks were completed. Functional correctness can be linked to bugs. Whenever a bug is introduced, the correctness is violated and when it is solved correctness is restored again. Functional appropriateness includes user experience, so it is not applicable.
- **Performance efficiency** is comprised of *Time behavior, Resource utilization* and *Capacity*. This characteristic is entirely not applicable, as it depends on performance at run-time, which we cannot measure when looking at static code.
- **Compatibility** contains *Co-existence* and *Interoperability*. This characteristic is also not applicable, as it is highly specific for a given system and very much depends on the use case.
- **Usability** is described by*Appropriateness recognizability, Learnability, Operability, User error protection, User interface aesthetics* and *Accessibility*. This entire characteristic looks at the user experience and is therefore also not applicable.
- **Reliability** consists of *Maturity, Availability, Fault tolerance* and Recoverability. This characteristic can also only be observed at runtime and is therefore not suitable.
- **Security** takes into account *Confidentiality, Integrity, Non-repudiation, Accountability* and *Authenticity*. This is highly specific to a system as well and thus not applicable.
- **Maintainability** considers *Modularity, Reusability, Analysability, Modifiability* and *Testability*. This characteristic is one of the main focus points of this thesis. Many of the metrics discussed before can be applied to one of these subcategories and can be derived from the source code alone.
- **Portability** includes *Adaptability, Installability* and *Replaceability*. This characteristic is once again not applicable as it depends on the use case and relates to environmental factors which are different for each project.

2.2 Related Work

There exist already many tools and services that utilize the data from collaboration platforms such as Github to analyze code and gain insight into projects. Expanding on these ideas and enhancing such tools has become a topic of research, as is evidenced by the following projects.

An evaluation engine is presented in [4] which determines how much a developer has contributed to a project in order to fairly distribute a donation among the people who have worked on it. Since the goal was to analyze a wide variety of repositories, the engine was restricted to metrics that were available for all projects. In the end the following two metrics were used for every analysis: changes, which includes additions and deletions and the history, consisting of commits and merges. Furthermore, [4] provided an option to include issues and pull requests as additional information. Then, a formula was created to calculate the contribution based on these relatively simple metrics. However, it was pointed out that the evaluation engine lacked an analysis of the quality of a contribution, which will be a focus point of this thesis.

In a project about the quantification of development value, [36], Hezheng Yin *et al.* developed an algorithm called DevRank, which is inspired by Google's PageRank [3]. DevRank calculates the value of a function considering how often the function is called, and how much development time is spent on a function. Thus, each function gets scored, with the idea, that a function that is used often is more important. Additionally, they also consider what they call non-structural value, which includes development effort, which is not directly reflected in the code and thus cannot be simply evaluated by parsing through the code. To do this, they used a supervised machine learning model to classify commits and rate their importance. By the combination of both methods, they were able to create an effective way to rate development value.

In a recent study by Hsi-Min Chen *et al.* [7], an automated programming assessment system called *ProgEdu* was developed, that uses a code-quality evaluation scheme to assess student contributions to programming projects. Their research heavily focuses on application in the context of university programming project evaluation, with a special interest in detecting free-riders. ProgEdu analyzes code quality by assigning a status to the project after a commit and the change of status is documented. The implementation relies on continuous integration of the project, as the goal is to get the status of *BuildSuccess*, which is achieved when the new code meets the requirements for coding style. Then, the team and individual contributions are calculated from the total amount of submissions, together with the number of times a status was achieved and the amount of status transitions.

Another study was published in 2018 [1], in which Patricia Rücker de Bassi et al. present 20 quality metrics that can affect the quality the source code. They analyzed an open source project with these metrics and used them to determine the contribution of each developer. The metrics were grouped into the following four categories: complexity, inheritance, size and coupling and each contribution was evaluated based on its influence on four project code quality measures: maintainability, testability, reusability and understandability. Every commit was thus assessed on whether it increased, decreased or maintained the code quality metrics, and whether the contribution had a positive, negative or no influence on the overall code quality.

In a 2009 study by Eirini Kalliamvakou et al. [24] a contribution function was developed, that calculates the share of each contributor by combining the lines of code that were written by that developer with the so-called contribution factor function of that developer. For the contribution factor function, *actions* and *project assets* are defined as a measurable

unit of change on a project part that can be contributed to. Actions are weighted according to their importance on a project and whether they have a positive or negative impact. Depending on the project, different actions are prioritized and their weights adjusted accordingly. To decide on the weights, projects are clustered together with similar projects and the actions are evaluated per cluster. The cluster-specific weights for the actions can then be used in the contribution function to calculate the contribution of the developer. The model built in this research was integrated into the Alitheia Core platform [34], which offers automated software quality analysis for open source projects.

The 2015 study [26] by Jalerson Lima et al. looked at developer contribution assessment from the perspective of project and team leaders. The goal was to support leaders in the evaluation of developer contributions by providing metrics derived from the repositories. The following metrics were considered in this study: *code contribution*, which was measured by lines of code, *average complexity per method*, for which the cyclomatic complexity was recorded for every added and changed method. Additionally *introduced bugs* detected parts of the code that were related to a bug and which developer was the last to change that code and finally *bug fixing contribution* recorded the developers associated with committing bug fixes. In an empirical study, these metrics were tested and discussed with team and project leaders. Both code contribution and complexity metrics were seen as useful by the leaders, the bug related metrics, however, were not unanimously received well. Partly this can be attributed to the way these metrics were calculated, as the metric recorded more bugs produced by more experienced developers, who also produce more code and thus are associated with more commits in general. Also the bug fix contribution metric does not account for the different ways people work, as it counts the commits that belong to a certain bug, but it does not take the complexity into account, so a developer solving bug in small increments would have more commits and thus be rated higher than one who puts the entire fix in one commit. Nevertheless, the metrics were helpful in the evaluation of developer contribution and could save the leaders some time.

Another study from the perspective of project managers was conducted by Matheus Silva Ferreira *et al.* in 2020. The goal of [13] was to assist project managers with regards to risk management and people management. This task, however, created a need to evaluate developers and a set of metrics was established to measure developers work. In the end, 64 metrics were chosen and grouped into the categories *Quality, Contribution, Collaborative Work, Degree of Importance, Productivity* and *behavior*. The majority of them are calculated from the simple metrics lines of code and number of commits, suggesting their importance in the evaluation. Through interview with project managers it became clear, that choosing the amount and nature of evaluation metrics is highly subjective and depends both on the context and on the person.

A different approach to evaluate code contribution was presented by Michail Tsikerdekis in 2017. In this paper, [35] a ranking algorithm called Persistent Code Contribution (PCC) was developed that is used to balance contributions from new and senior developers. PCC includes both quantitative and qualitative measures and was created for crowd-sourced projects. The algorithm considers code at the level of characters and evaluates how often a piece of code was changed. It is assumed that good quality code does not change (often);thus, the algorithm considers more stable pieces of code to be of higher quality. It was shown that the PCC algorithm assesses the contributions well and combines a range

of metrics into a single ranking. It was remarked that the algorithm could be useful to assist in the distribution of revenue among the developers of a project.

2.2.1 Discussion

As evidenced by the studies above there is no single metric to evaluate developers contribution, but rather a set of metrics that considers different aspects of a developers effort. In [13] it became evident, that the importance of a given metric is valued differently depending on the person and the project. This is also supported by [7], which highlights certain aspects of educational projects, that would not be as important in projects outside of a university. In several studies ([1], [24], [13]) similar metrics were grouped together and in [24] different weights were attributed to the categories depending on the type of project. For the purpose of this thesis, project leaders or repository owners should have a say in which metrics are more important in their projects than others. Thus, they should be given an opportunity to assign values for the weights for different metrics. The approach of combining similar metrics to categories will also be useful for this project. It allows for using many metrics, which reduces the overall impact of any single metric, while keeping the weight assignment relatively simple.

Chapter 3

Design

3.1 Requirements

The application should be able to fulfill the following requirements:

1. Work with any link to a public GitHub repository
2. Calculate the contribution percentage for an optionally given time-frame or for the entire project up to the date of evaluation.
3. Allow the repository owner to choose weights for the available metrics to allow for customization and better adaption to different projects.
4. Present an overview of the overall contribution percentage, as well as a more detailed report per contributor with scores for the different metrics to allow for improvement and make the calculated percentage more transparent.

3.2 Assumptions

For the evaluation, a public GitHub repository is required. We take for granted that the repository has been organically grown over some period of time, that multiple commits exist and that there is more than one contributor for the project. We also assume that the commit messages are sensible and that the entire repository is written in the same programming language, such that the contribution in terms of LOC can be reasonably compared. Since the project owner or team leader bears the responsibility to choose the weights and metrics, we may trust that they are familiar with the project and the requirements for their specific goals. We expect them to prioritize the project's success over their own financial gain, *i.e.*, we entrust them to choose the metrics in such a way that they ensure the requirements of the project are met and not such that the evaluation produces the highest percentage for them. Likewise, we presume the contributors to work towards improving the project and not to maximize their profit.

3.3 Architecture

In figure 3.1 the time dependency of the contribution engine is shown with a sequence diagram. The user starts the analysis process (method analyzeRepository()) by providing the repository as well as the desired metrics and optionally a timeframe and a weight vector. The weight vector allows the project owner to individualize the evaluation of the project and to place higher importance on some metrics over others. If no weight vector is specified, all chosen metrics are valued equally. The program then, with the help of PyDriller, gets the necessary commit information. In the next step, evaluates this data according to the specified metrics and provides an overview of the scores. Now the engine can apply the bonus calculated in this evaluation and derive the overall contribution percentage for each person. In the end, both a more detailed per person report, as well as the overall percentage are returned to the user.

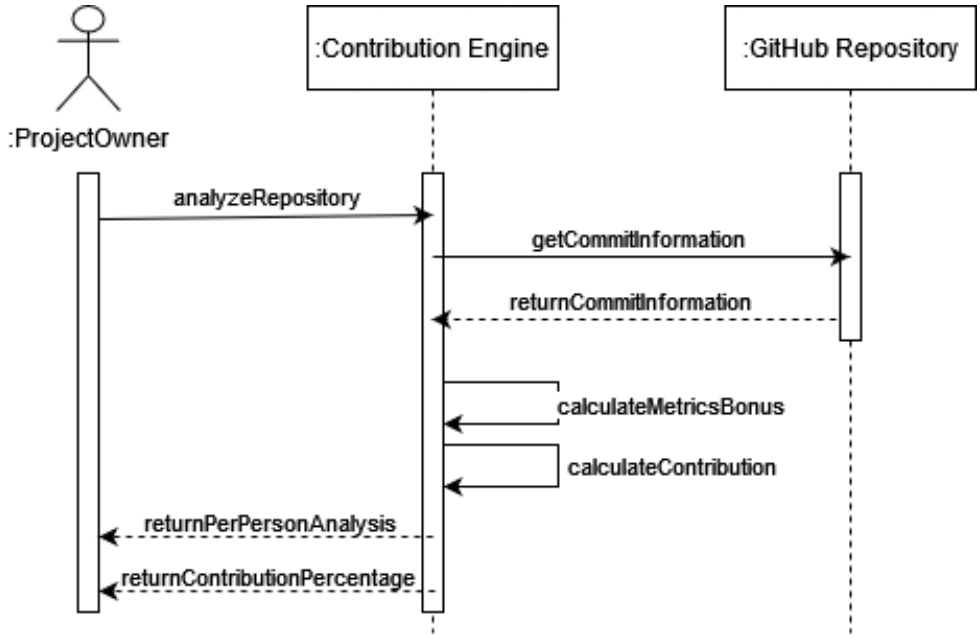


Figure 3.1: Contribution Engine Sequence Diagram

The evaluation process can be broadly split into the following four steps:

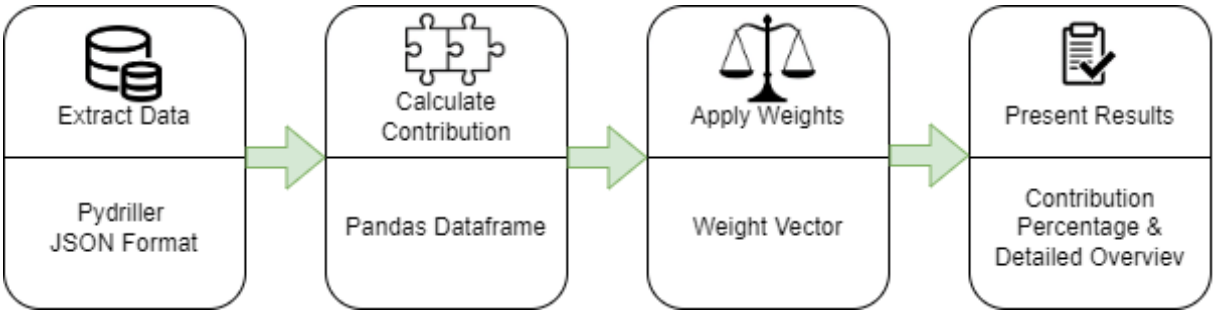


Figure 3.2: 4-step-process for contribution calculation of a GitHub Repository

- **Step 1: Extracting the data**: Before any analysis can be performed the relevant data must be accessed. For this purpose, the PyDriller tool is used. PyDriller is a Python framework for mining software repositories, which conveniently offers already some calculated metrics, such as complexity metrics and the lines changed in a commit. Since Pydriller uses its own *Commit object* class, which is cumbersome to work with, the commit information is converted to a dictionary during the data retrieval and consequently stored as a JSON file.

- **Step 2: Calculating the contribution percentage**: After extracting the information from the repositories, it can be further analyzed. In this step, the primary data is used to calculate metrics and evaluate contributions. For the evaluation, the JSON file is loaded into a Pandas dataframe, which makes further calculations easier. The scores are then combined in an overall contribution formula, which calculates every author in a repository their contribution percentage. This can then directly be used to calculate a developer's cut of a possible donation of a project.

- **Step 3: Introducing a notion of fairness**: No two projects are the same and not every project leader or repository owner values the same metrics. It is, therefore, important that the leaders have a say in deciding which metrics should be given more importance in the evaluation. To allow for this individualization the owner(s) of a GitHub repository will be able to assign weights to the different metrics, which were introduced in Step 2. To ensure that every developer is evaluated equally, the weights remain constant for an evaluation period.

- **Step 4: Presenting the results**: A developer contributing to a project should get some kind of feedback after an evaluation so as to allow them to improve their skills. However, it is important that the feedback encourages the creation of better code and does not cause the developer to just try to perfect all the metrics to get a better score. Though since many of the metrics are tied to code quality and not just quantity *i.e.,* lines of code, an improvement in the score should automatically be attributed to better code.

3.3.1 Contribution Calculation Formula

To calculate a total contribution of a developer, the amount of code is taken as the baseline, to which the rating with the different metrics is added:

$$C(d) = l + \alpha_1 a_1 l + \alpha_1 a_2 l \ldots + \alpha_n a_n l \tag{3.1}$$

Where C is the contribution, d the developer, l is the lines of code, $\mathrm{a_1}$ to $\mathrm{a_n}$ are the chosen metrics and α_1 to α_n are the weights the project leaders choose for each metric.

3.3.2 Choosing the metrics

As previously mentioned, it is important to include quality metrics when evaluating the contribution of a developer. This strategy is backed by several studies such as [1, 13, 24, 26]. While these studies use contribution quantity, which in most cases is measured in lines of code (LOC) as one of several metrics, they agree that the quality of a contribution matters as well. Collaboration metrics, such as comments on issues, and bug-related metrics, like introducing and fixing bugs, however, are not unanimously seen as beneficial and leave room for discussion. In interviews with team leaders and managers [26] found that some find the metric useful while others reject it.

For the purpose of this project, the following three metrics were chosen:

1. Delta Maintainability Metric
   This is one of the metrics that PyDriller can calculate. It is based on this so-called *Delta Maintainability Model* [2]. The model is used to determine how small code changes, such as a single commit, impact the maintainability of the overall project. *i.e.*, whether a commit has contributed to the deterioration of the project or whether it has maintained the quality standard and is a positive contribution to the project. The method assesses the added lines and categorizes them into beneficial and harmful change or high and low-risk change. The model considers three properties. The **Unit Size** is measured in LOC, **Unit Complexity** considers the McCabe cyclomatic complexity [28], which calculates the complexity of a program, and **Unit Interfacing** calculates the size of the interfaces by counting the number of interface parameter declarations. All three properties are measured in terms of *units*, which are defined as the smallest executable piece of code, i.e. methods. Every property has a threshold to categorize the units into low and high risk. The score for each property is then calculated by the formula 3.2.

   $$\text{dmm score} = \frac{\text{low risk change}}{\text{low risk change} + \text{low risk change}} \tag{3.2}$$

   To get the overall Delta Maintainability Metric, the average of the scores from unit size, unit complexity, and unit interfacing is calculated. By design of the Delta Maintainability Model, this dmm score is a number between 0 and 1, with 1 being

the perfect score and having only committed low-risk change. The bonus for this metric is then calculated as follows 3.3:

$$\text{dmm bonus} = \text{dmm score} \cdot \text{LOC} \tag{3.3}$$

These bonus lines are then added to the baseline LOC as described in formula 3.1. The overall contribution is then defined as the percentage of lines committed after applying the bonuses from all metrics.

2. Bug Fix Metric
While not everyone agrees on the benefit of this metric, it can still be useful for some project. In the context of this thesis, the Bug Fix Metric is considered as follows. It scans through the commit messages of each commit and checks for the keyword *fix*. If the message contains the keyword, a flag is set to 1, to mark the commit as containing a bug fix. All commits with the flag set to 1 will then count as a bonus for the overall contribution. Per default the bonus is set to $0.5 \cdot \text{LOC}$, a commit containing a bug fix is therefore considered 1.5 times as important as a non-bug fix containing a commit.

$$\text{bug fix bonus} = \begin{cases} 0.5 \cdot \text{LOC}, & \text{if bug fix} \\ 0, & \text{if no bug fix} \end{cases} \tag{3.4}$$

The factor can be adjusted later with the weight vector to give the metric more or less importance. This way solving bugs and restoring the project to a usable state is rewarded, but there is no bug-blaming metric, so making mistakes is not punished.

3. Test Coverage Metric
Having a high test coverage improves the reliability of a project [27]. Thus it is beneficial to have a metric, which rewards contributions to test files. This is exactly what this metric does. For every commit, it looks through the modified files to check whether a test file was changed and if that was the case, it sets a flag to 1. All commits that extended the tests will then also account for a bonus for the overall contribution. Similar to the bug fix metric, the bonus factor for a test file contribution is 0.5 per default.

$$\text{test coverage bonus} = \begin{cases} 0.5 \cdot \text{LOC}, & \text{if test file contribution} \\ 0, & \text{if no test file contribution} \end{cases} \tag{3.5}$$

The multiplication factor can here as well be adjusted with the weight vector if one desires to do so. Continuously working on the test files is thereby rewarded. Especially in the context of an educational project, where often a certain test coverage is expected, this metric can be useful.

It was decided to place the responsibility of choosing metrics on the project leader, as they are expected to know a lot about the project and have an overview of everything. They may choose which evaluation metrics to include based on whether they align with the project requirements or opt to omit other metrics if they are not applicable to a given project. As the choice of metrics should reflect the goals of the project as much as possible and should be based upon previously defined

requirements, the project leader should not have a hard time deciding whether a metric is to be included or not.

Every metric that is included in the evaluation puts a score on a commit and rates a certain aspect of a contribution. This scoring system can both be beneficial and problematic. On the one hand, it allows for comparison between commits, provides feedback for the developer, and assigns a numerical value to a contribution. On the other hand, however, it reduces a complex piece of work down to a single number, possibly losing some contribution information in the process. The more metrics are included in the evaluation, the less likely it is that a certain aspect of project contribution is completely omitted.

Now when it comes to defining the weights for the metrics, i.e. how much every metric should impact the final contribution calculation, one could decide to open up the decision to the community. While this would possibly ensure a higher acceptance for the weights, as it would be a collective agreement, it also has some drawbacks. If it is a bigger project with many contributors, many developers will not be familiar with the entire project and may focus more on some metrics that are more relevant to their part and devalue metrics that are not important to them, despite them possibly being beneficial for the bigger picture.

3.4 Implementation

In figure 3.3 a component diagram of the contribution engine is shown. The orange components are external classes, which are imported, but the green components are depending on them. It shows the relationship the user has to the two components he interacts with.

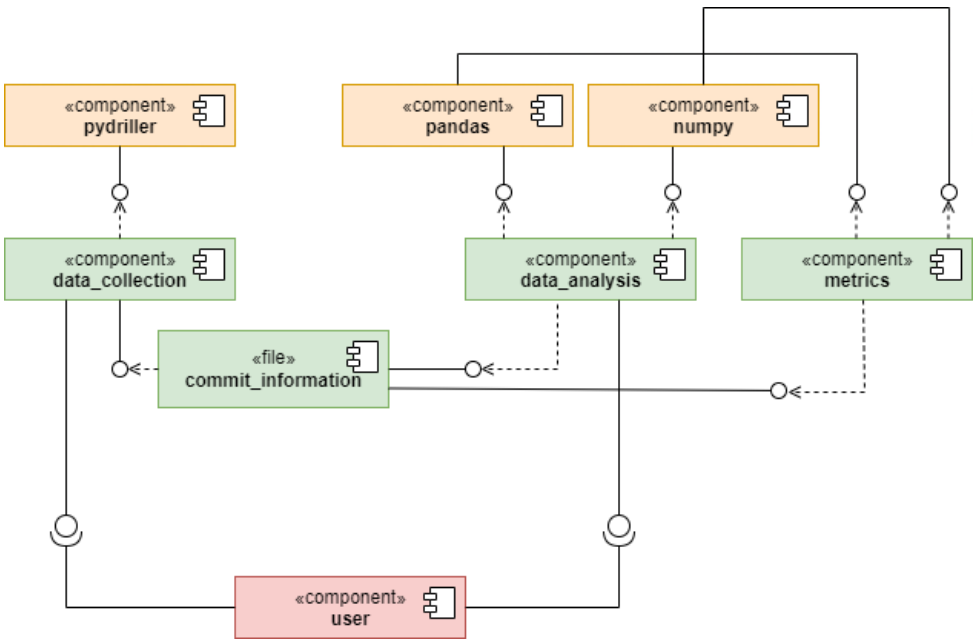


Figure 3.3: Contribution Engine Component Diagram

In the class diagram 3.4 a more detailed view of the classes is provided. As described in the workflow (3.2), the user must provide the URL to the GitHub repository they want to analyze. They may additionally specify a time frame by setting a *fromDate* and *toDate* if they do not want to analyze an entire project but are only interested in a shorter time span.

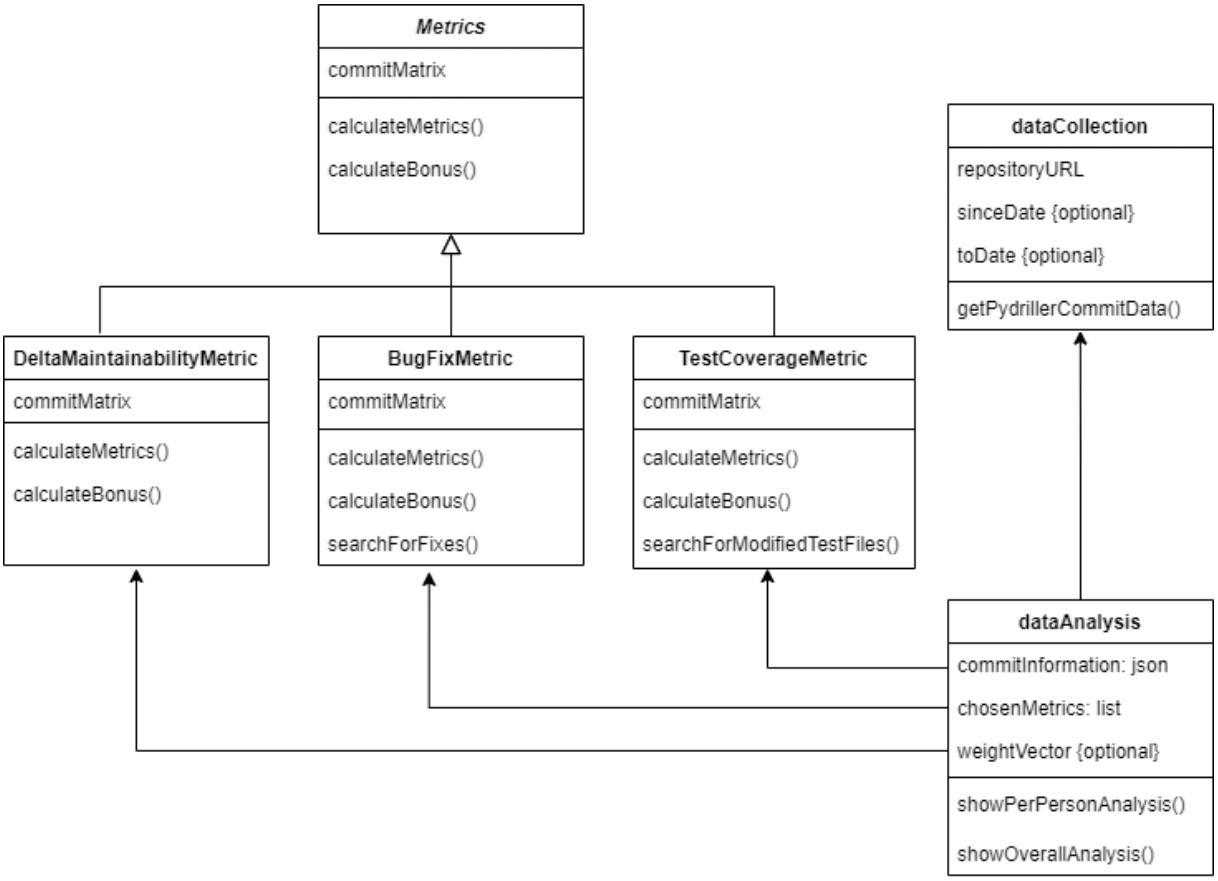


Figure 3.4: Contribution Engine Class Diagram

The *DataCollection* class now uses PyDriller to get the commit information and saves everything in a JSON file. For the *DataAnalysis* class, the user needs to choose what metrics should be used to calculate the contribution. The metrics are specified in a list. Optionally a weight vector can also be set, which allows the evaluator to value some metrics more than others and thus provides a more individual analysis.

The metric classes need to have two functions: *calculateMetrics()* is used to determine a score for the given metrics, which can be shown to the contributor as a means of personal analysis. E.g. for the bug fixes it counts the number of fixes while for the maintainability it displays the score between 0 and 1. *calculateBonus()* on the other hand is used to determine the bonus for the overall contribution. It can utilize the previously calculated metrics to derive the bonus per commit. Both score and bonus are stored in a Pandas dataframe.

After the metrics are calculated, the analysis continues. In *showPerPersonAnalysis()* the scores for each metric are displayed for each contributor. This analysis is meant to be

shown to the person, such that they can see how the overall contribution was calculated and may learn in which metrics they could improve. *showOverallAnalysis()* simply returns a contribution percentage for each person. This number can then be taken as the share of a donation every person should get. In this method, also the weight vector is applied to the bonuses, such that some metrics may give a higher bonus than others if the evaluator decides so.

Chapter 4

Evaluation

4.1 Repository 1 - SoPra Project

To begin the evaluation process, a known repository was chosen. This made it easier to assess the validity of the evaluation engine, as the project was already familiar. For example, it allows assessing whether selected metrics are able to represent a notion of fairness. In this case, the server repository of my Software Engineering Lab project [32] was used.

The repository considered in this first evaluation has 5 contributors, denoted *A*-*E*. Contributors *D* and *E*, which actually is the same person, with different accounts, have only contributed a small amount because they mostly worked on the client part of the project and only had a small contribution in the beginning in the server side.

4.1.1 Scenarios

We will evaluate three different scenarios against the baseline contribution with only LOC as a measure. They are illustrated with the diagrams in the next section. In the first scenario, as shown in Figure 4.1b, the project owner has not specified any additional weight but has decided to include all possible metrics (Delta Maintainability, Bug Fix, and Test Coverage).

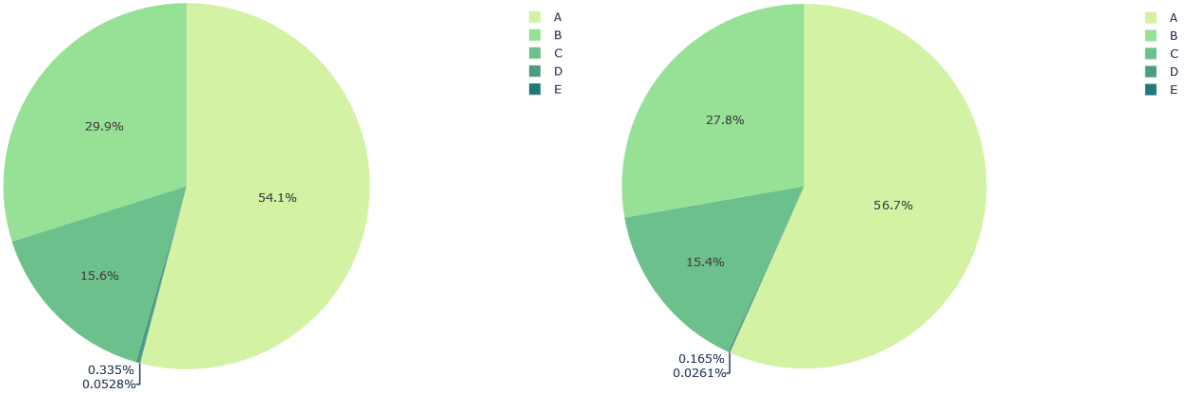
In the second scenario 4.2a, the project leader is adamant that the project is always deployable and that the product is constantly usable. They place high importance on fixing bugs, as this improves the usability of the prototype and will thus change the weights in order to favor the bug fix metric. In this specific example, the weight vector was $\begin{bmatrix}1 & 1 & 10 & 1\end{bmatrix}$ so the bug fix bonus is multiplied by 10 times, while the others (LOC, Delta Maintainability, and Test Coverage) remain the same.

In the third scenario 4.2b the test coverage is chosen as an important metric. As the project was developed in an educational context, where a percentage of test coverage was required, the project leader decided to reward any developer who works on improving the

test files. The weight vector was therefore set to $\begin{bmatrix}1 & 1 & 1 & 10\end{bmatrix}$ such that the test bonus is valued 10 times more highly than the other metrics.

4.1.2 Diagrams

As a baseline for the contribution percentage, LOC is used, as mentioned in formula 3.1. With no additional metrics and subsequently no additional weights either, the overall contribution looks as follows in Figure 4.1a. We can see, that person A was responsible for more than half of the lines contributed and contributor B committed two-thirds of the remaining part. The contribution of the users



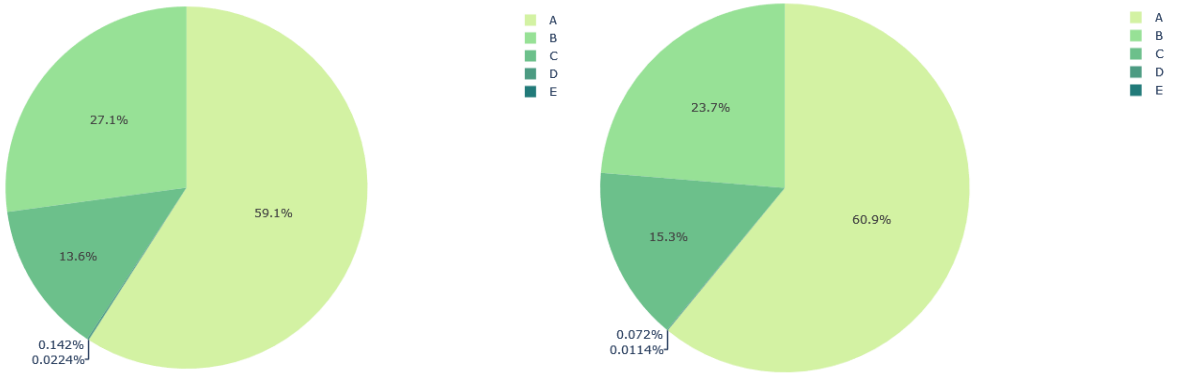
(a) LOC only (b) All metrics equal

Figure 4.1: Pie charts of contribution percentage when considering only LOC (left) and delta maintainability, bug fix, and test coverage metrics with equal weights (right).

If we now introduce the metrics, the distribution does not change significantly. In Figure 4.1b, we can see that A has a slightly higher percentage than before, while all other contributors lost some of their shares. However, we can also see that the change is only minor.

In the diagram of Figure 4.2a, it is possible to see that A has apparently fixed many bugs, as they are now responsible for almost 60 percent of the code committed. Person B has also solved a number of bugs, as their percentage has only shrunk by less than 1 % compared to when all metrics were considered equally.

In Figure 4.2b we see that contributor A has contributed a lot to the test coverage, as their percentage is over 6% higher than in the baseline LOC. Person C has evidently contributed more to test files than they have solved bugs, as they lost almost nothing compared to 4.1a. In contrast, developer B has lost about 4% of their contribution, indicating that they may have not been overly focused on writing tests.



(a) Focus on bug fix

(b) Focus on test coverage

Figure 4.2: Pie charts of contribution percentage with focus on bug fix (left) and focus on test coverage metric (right).

4.2 Repository 2 - Another SoPra Project

To have a fair comparison between two projects of relatively equal size, another SoPra project [22] was chosen as the second repository to evaluate. As both were for the same class, they had the same requirements and are thus built similarly.

This project had, however, ten different contributors (denoted *F*-*O*). But similar to the first project it has only 3 main contributors, as developers *K*-*O* have only contributed a minute amount.

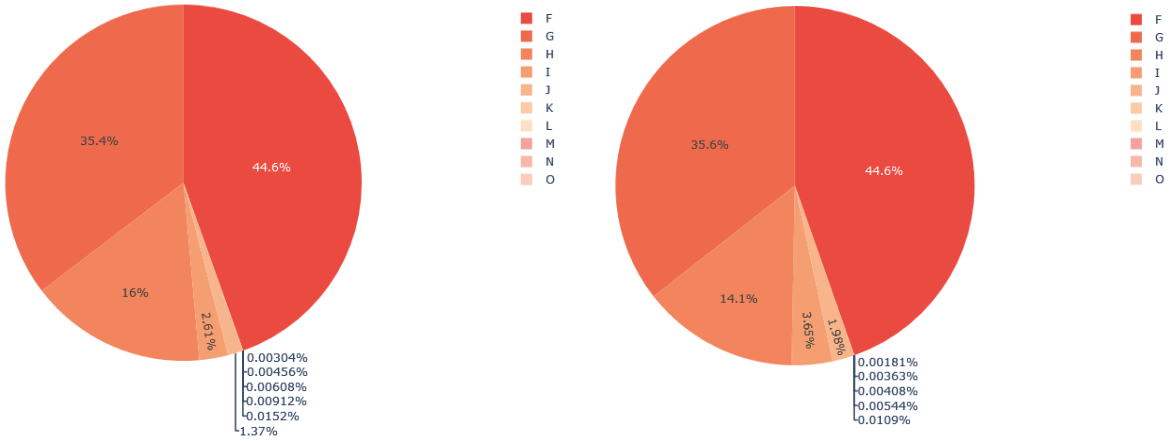
4.2.1 Scenarios

The scenarios for this evaluation are the same as before. So the baseline contribution is set with just LOC as a metric. Next, all available metrics are applied, but no weight is specified. Then the bug fix metric is given higher importance and finally, test coverage is highlighted.

4.2.2 Diagrams

Figure 4.3a again serves as the baseline contribution by just considering LOC as the only metric. We can observe that also in this project most of the work was done by three people, but unlike in the first project here, no single contributor is responsible for more than half of the code.

If we now introduce the metrics, almost nothing changes for developer *F*. They are still responsible for about 45% of the code. Contributor *H*, however, lost about 2% of contri-



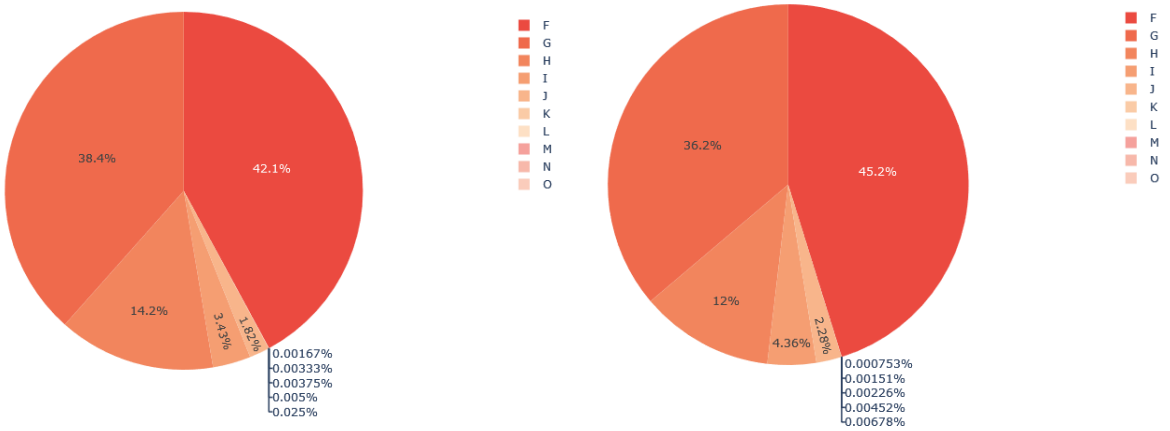
(a) LOC only

(b) All metrics equal

Figure 4.3: Pie charts of contribution percentage when considering only LOC (left) and delta maintainability, bug fix, and test coverage metrics with equal weights (right).

bution. In this repository also the smaller contributions of I and J cannot be ignored, as they both increase their percentage.

In Figure 4.4a the bug fix metric is highly valued. Developer G has evidently solved many bugs, as they gained about 5% of contribution here. Contributor F on the other hand lost a bit over 2% of their percentage. The contribution of H decreased a little compared to the baseline, but when comparing it to 4.3b it increased slightly. This indicates that H has fixed about the same percentage of all bugs as they have contributed in terms of LOC to the project.



(a) Focus on bug fix

(b) Focus on test coverage

Figure 4.4: Pie charts of contribution percentage with focus on bug fix (left) and focus on test coverage metric (right).

Finally in figure 4.4b the test file contribution is depicted. Noticeably here every contrib-

utor except for H increases their share. Apparently, person H did not prioritize writing tests and thus loses 3% of contribution.

4.2.3 Contribution over time

In the two evaluations before, the repositories were analyzed over their entire existence. Depending on the donation plan of a possible supporter, one might also be interested in a shorter timeframe. For example, if an investor decides to sponsor a project over a longer period of time, a monthly payment plan could be beneficial, as it gives the contributors a monetary reward for continuously working on the project. The implementation allows setting a start and end date for precisely this reason.

If we analyze project 1 under this aspect, the percentages look a bit different as compared to before. As a reference, we can see the daily contribution in terms of LOC per contributor over the entire project in Figure 4.5.

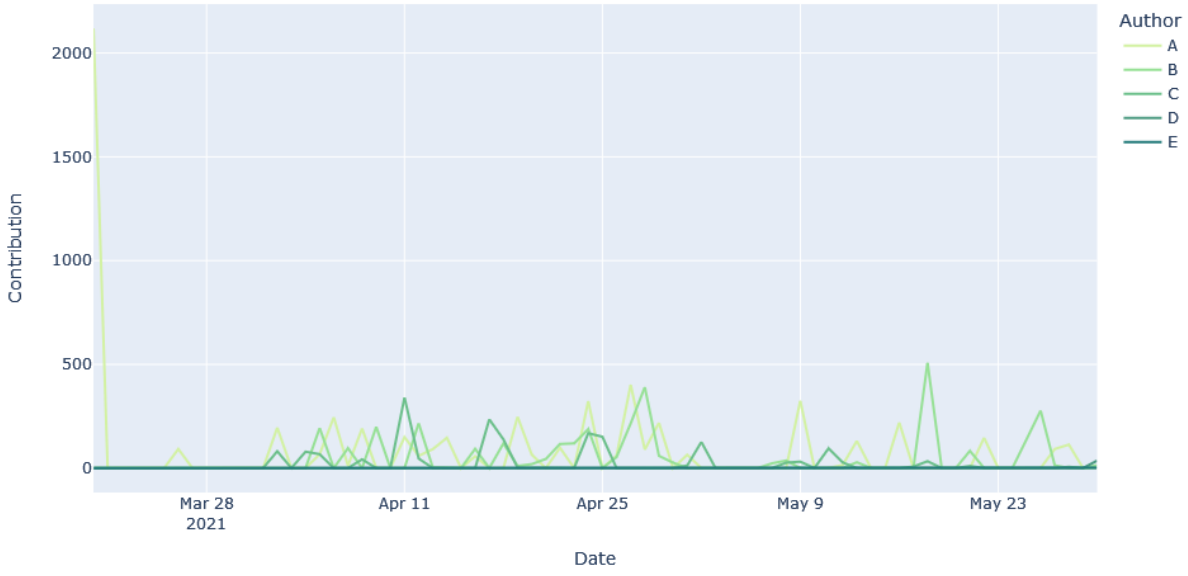
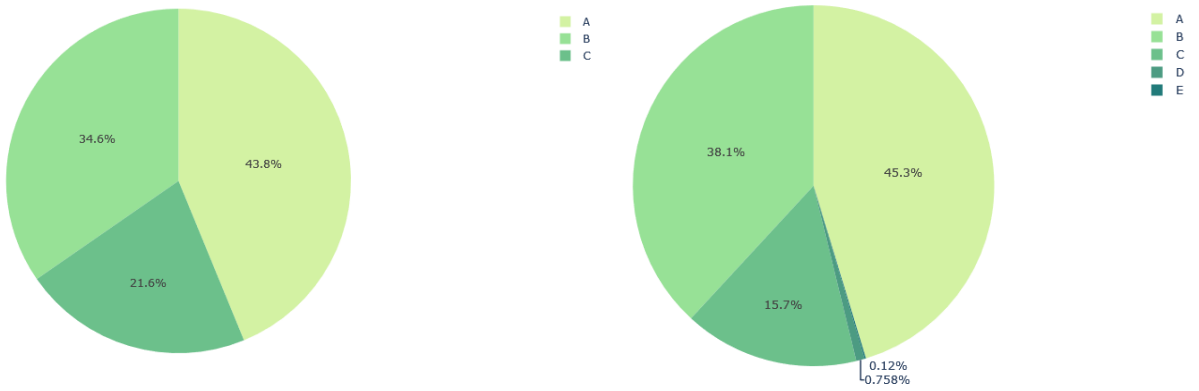


Figure 4.5: Contribution in terms of LOC of each author against time.

It shows that the three main contributors A, B and C were all involved during the entire process. In figure 4.6, we can now observer a clearer distribution. In April only developers A-C committed any work while in May all five people were involved. What immediately stands out is the fact that A has contributed less than 50% in both months, when before they were over that threshold in any scenario described above. When looking at 4.5 it becomes clear that this is the case, because A has a commit with a large amount of code that was committed in March, which was the initial setup of the project. Clearly, this impacts the overall contribution a lot, as this one commit contained 4 times as many lines as any other commit. It is also worth pointing out that C has contributed percentally more in April and would have received more than a fifth of the monthly budget, while in

the overall contribution visualized in 4.1b they get only about a sixth, which aligns with the percentage for the month of May.



(a) April 1 to April 30

(b) May 1 to May 31

Figure 4.6: Contribution percentage when considering all metrics equally over a monthly timeframe.

4.3 Per-Person Analysis

While the overall analysis and contribution percentage calculation are meaningful for a project leader when trying to distribute a donation among the contributors, it reduces the contribution down to a single number. To get a better understanding of how the percentage is calculated and to see how well they perform in each metric, a developer will likely be interested in a more detailed analysis of their work. The contribution engine provides this feature as well. For Repository 1 the per-person analysis looks as depicted in Table 4.1, which shows the scores for each metric and each person. The delta maintainability metric (dmm) is split into the three sub-metrics that were described in chapter 3, to display more clearly where the developer could improve.

Author	dmm unit size	dmm unit complexity	dmm unit interfacing	bug fixes	test coverage
A	0.323624	0.451220	0.424368	17	27
B	0.445292	0.577778	0.548337	16	24
C	0.559909	0.659091	0.652476	2	14
D	0.000000	0.000000	0.000000	0	0
E	0.000000	0.000000	0.000000	0	0

Table 4.1: Per-Person analysis of Project 1

Every developer will get the information of their line and can, thus, check how they performed and which metrics they could improve in. *C* for example would deduce that they could put more effort into fixing bugs, while *A* would probably decide to enhance their score for the unit size. The table shows the raw metric scores, so with no weight vector factor applied. Depending on the goals of the project, improvement in some metrics

will have more impact on the overall contribution percentage, but for feedback on their work quality, the unadulterated scores are chosen.

4.4 Discussion

One of the goals of this thesis was to extract metrics and indicators of a contributor's performance from a GitHub Repository. This was achieved with the use of PyDriller as a tool to extract the Delta Maintainability metric, as well as lines of code, and to analyze the commit messages. The second goal, defining a fairness concept to calculate the importance of a contribution with said metrics, was approached with the possibility to adjust the weights of the metrics.

4.4.1 Metrics

In the context of this thesis, the metrics maintainability, bug fix, and test file contribution were chosen as a means to judge the importance of a contribution. It was also planned to include information from the GitHub Issues, such as labels, comments, and numbers of issues closed per person. Open issues of a project can easily be extracted via the GitHub API, but closed points count as so-called Issue Events, and those can only be accessed if they are not older than 90 days [12]. This poses a problem.

Firstly, the aim of the contribution engine is to analyze a project over a given timeframe, or over the whole creation process to calculate the contribution of each developer, and finally their share of a possible donation. It would enforce a payment plan for continuous supporters at least quarterly or more frequently and would not allow a project to be evaluated in its entirety if it is older than 3 months.

Secondly, the example projects the contribution engine evaluated and was tested against are older than 90 days. It was, however, important to inspect a project that was well known, such that an informed decision could be made as to which metrics to include and which associated weights to choose.

The metrics were also chosen with the specific projects in mind. Especially the bug fix and test file contribution were analyzed with higher weights attached, as in the context the projects were created, these were highly valued. Both projects were part of a university course and thus arose in an educational context. It was important for the project to be constantly deployable and to ensure that the build does not fail, hence the higher importance of the bug fix metric. Additionally, a certain percentage of test coverage was required, therefore adding to the test files and thereby increasing the coverage was beneficial and is rewarded with the bonus from the test coverage metric. Still, as all three of the chosen metrics were important for the projects, hence why they were chosen, the fairest evaluation was the one where all metrics were valued equally.

Furthermore, the way the bug fix and test coverage metrics are currently determined leaves them vulnerable to manipulation. Since the bug fix metric searches for the keyword *fix*

in the commit message, any developer could just include this word in all their commit messages to get the bug fix bonus added to their line contribution. Likewise, for the test file contribution, the metric counts any change within a test file as a contribution towards test coverage, so one has to simply modify something in a test file to get this bonus. While this is not a problem for the hindsight evaluation done in this thesis, where the projects were already finished, this could be exploited when a project gets evaluated monthly and the developers know how the bonuses are calculated.

4.4.2 Fairness

As already described in chapter 2, fairness is not straightforward to be defined. In the context of this thesis, a concept of fairness was introduced with the possibility to choose which metrics to use from a list of possible metrics and adjust the impact of every chosen metric via the weight vector. In theory, this allows for a highly personalized evaluation that can be modified to suit different projects and match the requirements for every one of them. In practice, it turned out to be more complicated. To be able to set the weights and choose the metrics, one must be familiar with the project to ensure a reasonable evaluation. This means, that while the project does provide the opportunity to automatically calculate a contribution percentage, it still relies heavily on human input. How the project is currently designed will always require a human-in-the-loop to make informed decisions about the metrics and weights. While it would be possible to standardize the impact of every metric by analyzing many projects and using that information to define a baseline evaluation, *i.e.*, define a standard evaluation scheme, where the weights are all 1, we would lose the individualization.

Artificial intelligence could help to calculate the metrics. AI has already been used to analyze code quality [33], or to do performance testing [19], which respectively relate to the maintainability and test coverage metric of the contribution engine developed in this thesis. However, when it comes to setting the goal for a project, it must come from the project initiators. Therefore, a machine learning model may be useful in fine-tuning the metrics and possibly even in finding more metrics, but the overall goal or the information on which metrics should be optimized must be provided by a human. This is congruent with state-of-the-art research [18], where the benefits of using automated methods for code review are stated [23], but it is also highlighted that the human aspect cannot completely be omitted [6, 10],. If we define fairness in this context, *i.e.* a fair evaluation for a given project rewards any contribution that added to the project's overall success and possibly punishes contributions that hinder the progress, the input of what constitutes a successful project must come from outside and cannot be determined via analysis of the code. We can thus conclude that a purely quantitative approach is not enough to provide a reasonable notion of fairness and human involvement is a necessity.

Chapter 5

Final Considerations

5.1 Summary

To mitigate the problem many open source projects face in today's world, the likelihood of becoming deprecated or being abandoned because their key developers can not afford to work on them anymore, various platforms have been founded to allow projects to get sponsored and contributors to be financially rewarded for their work. This thesis presented a contribution engine to assist in a fair distribution of funds depending on much each developer has contributed to a project. To encourage good quality code, different metrics were used to evaluate the source code and to produce a bonus for well-performing developers. As a concept of fairness, a weight vector for the chosen metrics can be specified, such that the evaluation can be adjusted for different projects' needs.

Three metrics, maintainability, bug fixes, and test file contribution, were chosen to evaluate a known university project. The metrics were extracted from the GitHub Repository with the help of PyDriller and then analyzed with varying weight vectors to show how different priorities of metrics affect the distribution. A second, unfamiliar university project was used as a comparison, because the two projects were from the same course and thus subject to the same requirements, so it made sense to analyze them with the same metrics.

The first project was then also analyzed over a shorter timeframe to showcase the possibility of a monthly evaluation in case a sponsor would like to support a project over a longer period of time and periodically donate to it every month. This highlights how the contribution behavior changed for every contributor over time and permits a more detailed analysis.

Finally, a detailed per-person analysis was introduced, which can be shown to the contributors, such that they can comprehend how the overall contribution percentage was calculated and gives them a feedback on how they did so that they can improve certain metrics for the next evaluation period.

5.2 Conclusions

As already hinted at in chapter 4, the research goals have only partially been achieved. The first problem, *How to extract metrics or indicators of contributors' performance from an open-source Git repository*, was solved with the help of PyDriller. However, it was initially planned to get some additional information, specifically about GitHub issues with the help of the GitHub API. However, such an approach was not feasible because information about closed issues is only retrievable with the API when the issue has not been closed for more than 90 days and all issues in both projects, that were evaluated, were older than that. Therefore, it was not possible to include issue-related metrics in the evaluation, which could have been helpful indicators.

A considerable effort has been invested to retrieve the information from the GitHub issues, and when it did not work out due to the aforementioned reason, it was decided to focus on the metrics available via PyDriller. The calculation of the bug fix metric was originally planned to be done with the help of the issue label *bug*, as was the test coverage metric with the respective *test* label. As mentioned in the discussion of chapter 4, with the current implementation a developer could try to cheat the evaluation by specifically crafted commits that trigger the criteria to get bonuses. It is therefore necessary to find a compromise between not showing the developers exactly how the percentage is calculated and still giving them enough feedback on their work that they can improve.

The second research question, *How to define a fairness concept to calculate the importance of contributions in Git repositories based on metrics and indicators*, was answered with the introduction of the weight vector. While the vector with every metric valued equally was determined as fair in the projects evaluated, it can not be concluded that the same metrics and weight should be used to analyze different projects, as it has been customized to these specific projects. The contribution engine fairly evaluates the code contribution of SoPra projects, as long as the requirements are similar. However, to get a more general set of metrics, that are applicable to a wide variety of projects, more data and inside information from various projects would be needed, which would have been outside the scope of this thesis.

The requirements described in chapter 3 were fully met by the implementation. The application works with any URL to a public GitHub repository. The contribution percentage is by default calculated for the entire duration of the project, but can be adjusted if desired. The user can specify which metrics should be included in the evaluation and can change the weights of those metrics. They are finally presented with an overall contribution percentage for every developer as well as a per-person report with the calculated score for every metric.

When comes to the proposed and actual work schedules, there were differences. Most notably, the planned timeline suggested a linear approach with different project phases that are to be finished sequentially, while in practice most phases were done simultaneously or alternating, and there was always a need to adjust some parts, implement something, then change the design again and in the meantime also writing down the process. Especially once the API part did not work out, some steps had to be retaken and an alternative solution had to be found.

Defining a concept of fairness was much harder than anticipated, as it is not a quantifiable metric. Furthermore, it became clear that it was not possible to analyze one repository and generalize the findings to other projects. To assess whether an evaluation is fair for a given project, one needs to be familiar with said project. This was a limiting factor for this thesis, as the evaluated project was the only known one and thus the findings could not be applied to other projects.

5.3 Future Work

There are two major improvements the contribution engine could benefit from. Firstly metrics from many different types of projects could be used to create a more robust and generally applicable evaluation scheme. Machine learning algorithms could possibly be used to analyze more projects and find new metrics or indicate beneficial contributions. It would also be nice if non-code-related metrics, such as writing requirements or assigning and commenting on issues, could be included in the evaluation, as they also contribute to the success of a project, but are not directly reflected in the commits.

Secondly, a graphical interface would be useful. Especially once more metrics are considered, it would provide a better overview and be more intuitive to use, if a user can just click the desired metrics and directly adjust the weights, rather than writing out a python list. Also, for the per-person analysis and the contributor feedback, it would be nice to have some graphics and a pretty score display instead of a table of numbers.

Bibliography

[1] Patricia Rücker de Bassi et al. "Measuring Developers' Contribution in Source Code using Quality Metrics". In: *2018 IEEE 22nd International Conference on Computer Supported Cooperative Work in Design ((CSCWD))*. 2018, pp. 39–44.

[2] Marco di Biase et al. "The Delta Maintainability Model: Measuring Maintainability of Fine-Grained Code Changes". In: *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. 2019, pp. 113–122.

[3] Sergey Brin and Lawrence Page. *The Anatomy ofa Large-scale Hypertextual Web Search Engine*. 1998. URL: http://infolab.stanford.edu/pub/papers/google.pdf.

[4] Falter Calvin. *Design and Implementation of a Commit Evaluation Engine of an Open Source Donation Platform*. Binzmuehlestrasse 14, 8050 Zurich, Switzerland, Nov. 2020.

[5] Chris Ceplenski. *Employee Rewards: The Importance of Perceived Fairness*. 2013. URL: https://hrdailyadvisor.blr.com/2013/06/29/employee-rewards-the-importance-of-perceived-fairness/.

[6] Marco Cerliani. *Quality Control with Machine Learning*. 2019. URL: https://towardsdatascience.com/quality-control-with-machine-learning-d7aab7382c1e.

[7] Hsi-Min Chen, Bao-An Nguyen, and Chyi-Ren Dow. "Code-quality evaluation scheme for assessment of student contributions to programming projects". In: *Journal of Systems and Software* 188 (2022), p. 111273. URL: https://www.sciencedirect.com/science/article/pii/S0164121222000358.

[8] Sarah Conway. "The Linux Foundation launches new CommunityBridge platform to help sustain open source communities". In: (2019). URL: https://www.linuxfoundation.org/press-release/the-linux-foundation-launches-new-communitybridge-platform-to-help-sustain-open-source-communities/.

[9] Devon D. Zuegel. "Announcing GitHub Sponsors: an new way to contribute to open source". In: *Github Blog* (2019). URL: https://github.blog/2019-05-23-announcing-github-sponsors-a-new-way-to-contribute-to-open-source/.

[10] Unicorn Developer. *Machine Learning in Static Analysis of Program Source Code*. 2020. URL: https://medium.com/pvs-studio/machine-learning-in-static-analysis-of-program-source-code-21bdc7e8ce6d (visited on Sept. 7, 2022).

[11] Cambridge Dictionary. *fairness*. URL: https://dictionary.cambridge.org/dictionary/english/fairness (visited on Sept. 7, 2022).

[12] *Events API*. URL: https://docs.github.com/en/rest/activity/events (visited on Sept. 7, 2022).

[13] Matheus Silva Ferreira et al. “How is the Work of Developers Measured? An Industrial and Academic Exploratory View”. In: *Journal of Software Engineering Research and Development* 8 (2020), pp. 1–20. URL: https://sol.sbc.org.br/journals/index.php/jserd/article/view/544.

[14] Daniel M. German et al. “”Was My Contribution Fairly Reviewed?” A Framework to Study the Perception of Fairness in Modern Code Reviews”. In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 2018, pp. 523–534.

[15] *git*. URL: https://git-scm.com/ (visited on Sept. 7, 2022).

[16] *GitHub*. URL: https://github.com/open-source (visited on Sept. 7, 2022).

[17] Kevin Grossman. *The Fairness Perception Problem Plaguing Recruiting*. 2022. URL: https://www.ere.net/the-fairness-perception-problem-plaguing-recruiting/.

[18] Anshul Gupta. “Intelligent code reviews using deep learning”. In: 2018.

[19] Mahshid Helali Moghadam et al. “Machine Learning to Guide Performance Testing: An Autonomous Test Framework”. In: *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2019, pp. 164–167.

[20] Synopsys Inc. *2022 Open Source Security and Risk report*. 2022. URL: https://www.synopsys.com/blogs/software-security/open-source-trends-ossra-report/.

[21] ISO/IEC. “ISO/IEC 25010 (2011) Systems and Software Engineering -Systems and Software Quality Requirements and Evaluation (SQuaRE) -System and Software Quality Models.” In: (2011). URL: https://www.iso.org/standard/35733.html.

[22] *Just One*. URL: https://github.com/sopra-fs-20-group-15/server (visited on Sept. 7, 2022).

[23] Sylvain Kalache. *The future of code quality, security and agility lies in machine learning*. 2019. URL: https://www.cio.com/article/219673/the-future-of-code-quality-security-and-agility-lies-in-machine-learning.html (visited on Sept. 7, 2022).

[24] Eirini Kalliamvakou et al. “Measuring Developer Contribution From Software Repository Data.” In: Jan. 2009, p. 55.

[25] Frank J. Landy, Barnes Janet L, and Kevin R. Murphy. “Correlates of Perceived Fairness and Accuracy of Performance Evaluation”. In: *Journal of Applied Psychology* 63.6 (1978), pp. 751–754.

[26] Jalerson Lima et al. “Assessing developer contribution with repository mining-based metrics”. In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2015, pp. 536–540.

[27] Y.K. Malaiya et al. “The relationship between test coverage and reliability”. In: *Proceedings of 1994 IEEE International Symposium on Software Reliability Engineering*. 1994, pp. 186–195.

[28] T. McCabe. “Cyclomatic complexity and the year 2000”. In: *IEEE Software* 13.3 (1996), pp. 115–117.

[29] Frank Nagle et al. *Census II of Free and Open Source Software — Application Libraries*. Harvard Laboratory for Innovation Science (LISH) and Open Source Security Foundation (OpenSSF), 2022. URL: https://linuxfoundation.org/tools/census-ii-of-free-and-open-source-software--application-libraries/.

[30] Molly B. Pepper and Seemantini Pathak. "Classroom Contribution: What Do Students Perceive as Fair Assessment?" In: *Journal of Education for Business* 83.6 (2008), pp. 360–368. URL: https://doi.org/10.3200/JOEB.83.6.360-368.

[31] Song Qin. *Flatfeestack Github Reposotory*. 2022. URL: https://github.com/flatfeestack (visited on Sept. 7, 2022).

[32] *SoPra FS21 - Group 05 Server - Pictures Game*. URL: https://github.com/sopra-fs21-group-05/group-05-server (visited on Sept. 7, 2022).

[33] Sudeep Srivastava. *AI in Quality Assurance: The Next Stage of Automation Disruption*. 2022. URL: https://appinventiv.com/blog/ai-in-quality-assurance/ (visited on Sept. 7, 2022).

[34] *The home of Alitheia Core*. URL: https://sqo-oss.org/ (visited on Sept. 7, 2022).

[35] Michail Tsikerdekis. "Persistent Code Contribution: A Ranking Algorithm for Code Contribution in Crowdsourced Software". In: 23.4 (2018), pp. 1871–1894. URL: https://doi.org/10.1007/s10664-017-9575-4.

[36] Hezheng Yin. "Quantifying the Development Value of Code Contributions". MA thesis. EECS Department, University of California, Berkeley, 2018. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-174.html.

List of Figures

List of Tables