University of Passau

Department of Informatics and Mathematics

Master's Thesis

On the Influence of Developer Coreness on Patch Acceptance: A Survival Analysis

Author:

Christian Hechtl

March 16, 2020

Advisors:

Prof. Dr.-Ing. Sven Apel

Chair of Software Engineering I

Prof. Dr. Gordon Fraser

Chair of Software Engineering II

Thomas Bock

Chair of Software Engineering I

Hechtl, Christian:
On the Influence of Developer Coreness on Patch Acceptance: A Survival Analysis
Master's Thesis, University of Passau, 2020.

Abstract

The influence of a developer's reputation within an open-source software (OSS) project on the success of said developer's contributions to the project is not fully known. Since it is a key objective to gain a good reputation within an OSS project for voluntary developers, this is an interesting research objective. For that reason, we replicate a study by Bosu et al. [BC14] on this topic. They find that developers with a higher reputation are more likely to receive a faster first-feedback on their contributions and that they have a lower review interval in general until the acceptance. Moreover, they find that developers with a higher reputation are more likely to get their contributions accepted. In a previous study we were not able to confirm these results. In this thesis, we try to replicate the results by analyzing the data of nine different OSS projects using survival analyses.

We are again not able to confirm the results by Bosu et al. We find that the success of a contribution seems to be dependent on other factors than the reputation of a developer. The success of a contribution seems to be very project specific and thus is influenced by different factors for different OSS projects. Developers with a higher reputation do, in general, not seem to have a shorter first-feedback interval, a higher feedback rate, a shorter review interval, a higher code-acceptance rate, and do not need less revisions of a contribution than developers with a lower reputation.

Contents

List of Figures

List of Tables

1 Introduction

Open-Source Software (OSS) development is an ever growing field of modern day software engineering. Some OSS projects dominate complete aspects in our technologically connected world like Linux is the only used operating system on the top 500 supercomputers in the world.[1] Nevertheless, most of the work in OSS projects is done by volunteers, but why is that? Lakhani et al. [LW05] find that one of the main reasons for this is that developers seek a good reputation within such projects. This reflects a common social structure in groups of people.

One socio-technical question that arises from this is whether a good reputation influences the outcome of the contributions made by a developer to an OSS project. The outcome of such a contribution is dependent on the review process of an OSS project in which the contributions is reviewed and checked for flaws or possible enhancements. Bosu et al. [BC14], investigate said question by conducting an empirical study based on the data of OSS projects and find, that a better reputation of developers indeed has a positive influence on their contributions. They find that this influences the success and duration of the review of a submission. Furthermore, they find, that there is also an influence on the time span until a developer gets a first feedback on a contribution. In a previous study [Hec18], we tried to recreate and confirm these results. We did this by analyzing three OSS projects that use a different contribution system than the projects Bosu et al. investigate. Moreover, we used a different metric to measure the reputation of a developer. We were not able to confirm that developers with a higher reputation have a higher success rate for their contributions than developers with a lower reputation within an OSS project.

The goal of this thesis is to further study whether the reputation of developers has an influence on the outcomes of the review of their contributions to OSS projects. To do this, we use a different approach than before. In many previous studies, including our own, on this topic, the researchers look at the projects as a whole from beginning to now. We, on the other hand, split the social network into 9-month ranges and analyze these separately. We do this because there is a lot of

[1] https://www.top500.org/statistics/details/osfam/1

change in contributors during the lifespan of a OSS project. The case studies we analyze for this thesis are Jailhouse, BusyBox, FFmpeg, GCC, Git, LLVM, QEMU, U-Boot, and FLAC, for which we obtain the data by mining version-control systems and mailing-list archives. Moreover, most previous studies, like our previous study, represented the reputation of a developer by classifying them into core and peripheral developers. They then assume that core developers have a higher reputation. In this thesis, we do not classify the developers into groups but measure their *coreness* by applying two different network metrics. We do this because there is no unified approach to classify developers into the two groups and thus there is a lot of bias in the selection of the method involved that can lead to corrupted data. Subsequently, we map mails to patches to be able to analyze the patches on the mailing-lists of the OSS projects, since we only analyze projects that use a mailing-list as contribution-tool. Finally, we apply Survival Analyses and other Regression Models to the data, to check whether there is an influence of the *coreness* of developers on their contributions.

We are not able to confirm the findings of the original study. They find that developers with a higher reputation have a shorter first-feedback interval, a shorter review interval, and a higher code-acceptance rate than developers with a lower reputation within an OSS project. We are not able to confirm these findings, as we see that this seems to differ from OSS project to OSS project. The influence of the reputation on the outcomes of the review process of a contribution seems to be dependent on other factors than just the reputation of a developer. As for the number of revisions a developer has to send until the acceptance of a patch, we are not able to show that this number is lower for developers with a higher reputation within an OSS project. This follows the findings of the original study.

The rest of the thesis is structured as follows: In Chapter 2, we give an overview of topics related to this thesis, including details acout OSS development, social-network analysis, survival analyses, and related work. In Chapter 3, we present the original study. In Chapter 4, we present the research question, the hypotheses, and the approach, we use for the analyses. In Chapter 5 we present the results of the analyses followed by a discussion. In Chapter 6, we present the threats to validity of this work. Finally, we draw a conclusion and give an outlook on future work in chapter 7.

2 Background

In the following chapter, we present background knowledge on topics related to this thesis. Namely, we present information about the open-source software (OSS) development process, social-network analysis (SNA), and survival analyses. Moreover, we present related work.

2.1 OSS Development

To be able to talk about influences on the OSS development process, we have to explain what this is. In the following we explain the details of this process.

2.1.1 The Patch-Review Process

One of the biggest upsides of OSS development is that any willing developer can contribute to it. However, this also brings the need for some kind of controlling instance to prevent bad contributions from getting integrated. For OSS projects, this is the patch-review Process. Asundi et al. describe this process to be similar to code-reviews or code-walkthroughs in closed commercial projects [AJ07].

Since contributors to OSS projects can be scattered around the globe, there has to be a mechanism in place that is not influenced by different time zones and distances. Zhu et al. identify two such contribution tools: *patch-based* and *pull-request-based* tools [ZZM16]. In this thesis we focus on OSS projects that use a mailing-list as contribution system. This falls under the category of patch-based tools which is why we only explain this type of contribution tool in detail.

In Figure 2.1 we illustrate the patch-review process for a project that uses a mailing-list. After cloning the code-base and working on changes, developers first need to submit these changes to the mailing-list. These submissions are also called *patches*. Said patches are then incorporated into the review process. Reviewers then have to check the patch for problems. Reviewers have three different ways to react to a patch. They can accept it, reject it, or request changes by leaving a review comment. In case there are requested changes, the original author now has to incorporate the

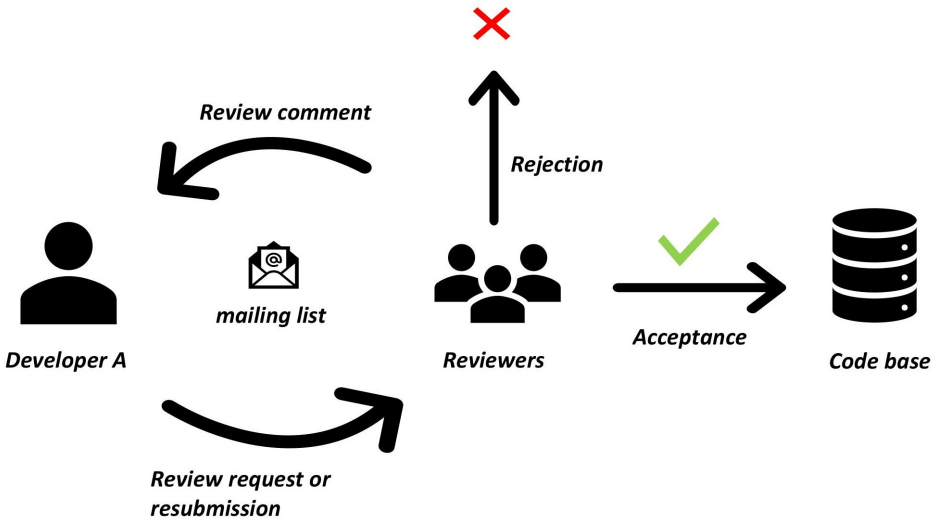


Figure 2.1: Patch-review process. Developer A originates a patch and submits it via the mailing-list to the reviewers. They either request changes, accept the patch into the code base or reject it completely.

requests and resubmit the changed patch, also called a *revision*. The set of all such revisions of one patch are called *revision set*. Once the revision set gets accepted, the changes are integrated into the code-base and the patch is successfully accepted.

2.1.2 Pros and Cons of the Contribution System

The review of contributions to OSS projects relies on other involved developers of the project, as we describe in Section 2.1.1. This brings some upsides and some downsides with it. Asundi et al. [AJ07] identify some of these. On the positive side, they state that contributing developers are usually very interested in the project. They argue that this is based on the fact that most of the contributors in OSS projects are volunteers. And since voluntary contributors can choose whatever project they like, they are usually keen on the project's success. Furthermore, the review is, in most cases, done by more than one developer. This very likely increases the probability of errors and other flaws being detected during the review process.

On the negative side, however, they state that the voluntary code review can lead to contributions being forgotten or taking a long time. This can happen if the contribution of a new developer is not reviewed right away and thus gets forgotten since no reviewer knows the developer. This can affect the motivation of the developer whose contribution is under review. With this, the potential of developers leaving the project arises.

Czerwonka et al. [CGT15], on the other hand, find only downsides to the review process of OSS projects. Although they review the process of commercial projects at *Microsoft*[1], their findings can be used as a proxy for OSS projects, since Microsoft uses the same kind of review process. Czerwonka et al. find that only about 15% of review comments find functionality errors. 50%, which represents the majority, are

[1]Microsoft homepage: https://microsoft.com

only concerned with the long-term maintainability of the project. This leaves room for speculation, whether or not the review process is as effective as it should be.

2.2 Social Network Analysis

In the following, we present the basics of social-network analysis (SNA). This is a central part of our study, because we use network analysis metrics to proxy the reputation of a developer. We do this to see whether the reputation has an influence on the outcomes of the review process.

A network is a construct that describes a relationship between people or things. Such networks are usually represented by graphs. The formal notation of such being $G = (V, E)$, with V describing the set of vertices and E the set of edges between the vertices. The relationship between two vertices is defined as an edge. Therefore, the set of edges is defined as $E \subseteq V \times V$. Moreover, an edge can be directed or undirected. Directed edges have a defined vertex as starting point and one vertex as endpoint. An undirected edge, on the other hand, is just a link between two vertices without a direction. Edges can also be simplified. In this process, multiple edges that represent a connection between the same two vertices are unified into one edge. This is usually done in terms of making the illustration of a graph easier. [BE05]

Network analysis is a common research approach used in many different fields of research, like electrical engineering, project planning, biology, and many more [BE05]. In our case we analyze social networks. This subclass of networks usually represent interactions between people. In the context of software engineering such networks usually describe interactions like mails between developers. Such networks are called *developer networks* [LFRGBH09], which we describe in the following.

2.2.1 Developer Networks

Joblin et al. [Job17] identify two base classes of developer networks: *coordination networks* and *communication networks*. These kinds of networks serve as a means to analyze social structures, like the reputation of a developer, in open-source software (OSS) projects.

Coordination networks are usually constructed from commit data, extracted from a project's version-control system. Such networks are called *co-change networks*, that describe the relationship between two developers based on changes done on the same file or piece of code.

Contrary to that, communication networks describe communication between developers. Such networks are usually constructed from public mailing-list archive data.

Such networks are sometimes constructed with the complete commit or mail data of an OSS project, meaning all the available data of the complete lifetime of a project. This brings on a problem in terms of generalizability. Foucault et al. [MF15] find that there is frequent developer turnover in OSS projects. Such turnover is quite common in OSS projects and can be a big problem for the project, because there are always new developers that need to familiarize themselves with the specifics of a project, which takes a lot of time. In this thesis, we use coreness, which we describe

in Section 2.2.2, as proxy for the reputation of a developer. This can be strongly affected by developer turnover. For this reason, we split our developer networks into subsequent ranges of nine months and analyze the eigenvector-centrality and hierarchy of these.

In this thesis we analyze both coordination and communication networks. In addition, we also analyze a combined network containing both co-change and mail edges.

2.2.2 Coreness in Social Networks

Since we analyze the impact of developer reputation on their contributions to OSS projects, we have to introduce how we measure the reputation. We use the coreness of a vertex, i.e., the coreness of a developer within the network, as a proxy for this. The coreness describes how central a vertex is in the network. To measure this, we use two different network metrics: *eigenvector centrality* and *hierarchy*.

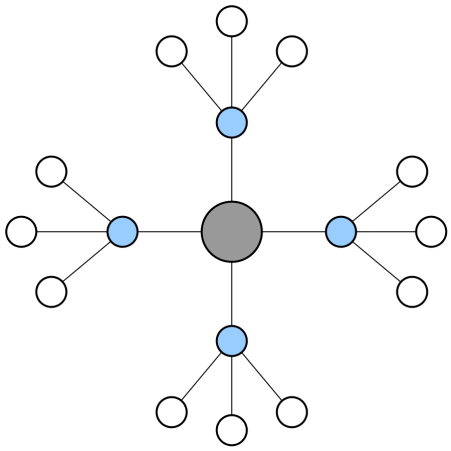


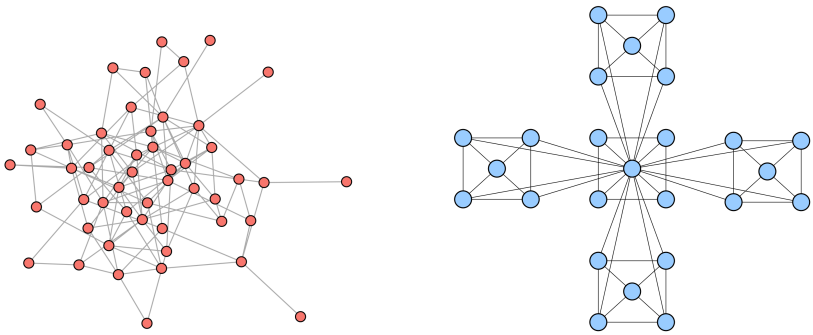
Figure 2.2: Small network example. The vertex with the highest eigenvector centrality is the gray one in the middle, since it is connected to 4 vertices with a degree of 4. The other ones are all connected to vertices with a lower degree. This figure is taken from [Job17] Figure 2.4.

Eigenvector centrality is a commonly used measure to determine how central a vertex is within a network. In detail this metric determines the centrality of a vertex within its local neighborhood in the network, meaning the centrality of a vertex is determined by the centrality of the other vertices in its direct neighborhood. The higher their centrality, the higher the centrality of the vertex under investigation. Formally, the centrality value x_i for each vertex i is calculated with the following formula:

$$x_i = \frac{1}{\lambda} \sum_{j \in N(i)} x_j \tag{2.1}$$

$N(i)$ represents the set of neighbors of i and λ represents a proportionality constant [JAHM17, BE05].

Hierarchy, on the other hand, is a more complex metric to determine the coreness of a vertex. Joblin et al. [JAHM17] describe this metric as a means to determine



Random Network Hierarchical Network

Figure 2.3: A random, unorganized network on the left and a hierarchical network on the right. The hierarchical network is organized in clusters, which are connected to each other and thus all together build a larger cluster. The top of the hierarchy is represented by the vertex in the middle because it exhibits a high degree and a low clustering coefficient. The figure is taken from [Job17] Figure 2.9.

how local neighborhoods are organized among each other. This is deduced by a combination of two other metrics: *node degree* and the *clustering coefficient* [RB03].

The node degree is the number of outgoing and incoming edges of a vertex. The clustering coefficient describes how embedded a vertex is within a cluster of vertices. Moreover, it gives a quantitative measure of the likelihood that the neighbors of a vertex are connected to each other [JAM17]. The formal definition of the clustering coefficient c_i for a vertex i is defined as:

$$c_i = \frac{2n_i}{k_i(k_i - 1)}, \tag{2.2}$$

k_i is the number of edges that are connected to i and n_i the number of edges between the neighbors in a cluster [BLM+06].

Once these values are determined, the hierarchy of a vertex can be extracted. The higher the degree and the lower the clustering coefficient of a vertex, the higher its hierarchy.

We define the coreness of a vertex as its eigenvector degree or its hierarchy value. The higher these values, the higher the coreness. We use these coreness-scores of developers as a proxy for their reputation within the OSS project, as described in the following section.

2.3 Survival Analysis

Survival analyses are a type of statistical analyses that originate in the field of medicine. They usually describe the probability of a patient surviving, given a certain period of time [RNP+10, LRS17]. In more general words, a survival analysis is usually used to analyze the time until an event occurs under the influence of some other variable, e.g., how long two groups of patients survive an illness, where one group is given a drug and the other is given a placebo. The time variable is usually called the *survival time*, while the event is called *failure*. This failure can either be

a positive or a negative event, e.g., the death of a patient or the full recovery of a patient. This depends on the objective of the analysis. [DGK12]

One key element of survival analyses is data censoring. Censoring of data occurs if the survival time of a subject is not exactly known. There are three main reasons why this can occur: the event does not occur for a subject before the study ends, the subject can no longer be traced during the study, or the subject drops out of the study. This kind of censored data is called right-censored data, because the true survival time of a subject is greater than or equal to the observed time. Since we are only concerned with right-censored data in this study, we only explain this type of censored data. [DGK12]

There are two key building blocks for every survival analysis: the *survivor function* and the *hazard function*. The survivor function denotes the probability of a subject "surviving", i.e., not experiencing the event past a specified time. The hazard function describes the probability of a subject experiencing the event of the study in a certain time unit, if the subject survived until a specified time. [DGK12]

Although these explanations are all based on survival analyses in the medical field, there is also a variety of other fields of research that use survival analyses, like in our case software engineering. In Section 2.4, we give an overview of studies in the context of software engineering, that use survival analyses.

In this thesis, we use two different survival measures: the *Kaplan-Meier method* and the *cox proportional hazard*. We describe these in the following.

2.3.1 Kaplan-Meier Method

The Kaplan-Meier method or Kaplan-Meier estimator is a method to estimate the survival probability of a subject at a given point in time. To do this, one first brings the data into the following form for every group of data, i.e. for all different values of the influencing variable:

failure times	**# of failures**	**censored**	**risk set**

Table 2.1: Data format for the Kaplan-Meier Method. Taken from [DGK12] page 60

The failure times here denote each points in the study time, starting at 0, where a failure happens. The number of failures is the number of failures that happen in a given time interval. Censored describes how many subjects were censored in the last time interval and the risk set is the number of subjects that are still "alive" at the given point in time. [DGK12]

Subsequently, one has to calculate the survival probability S for a given time t. We start with time 0. The probability here is always 1. The remaining probabilities are then calculated as follows:

$$S(t) = S(t-1) \times \frac{riskset(t)}{riskset(t) - failures(t))} \tag{2.3}$$

Once this probability is calculated, one has to apply the *log-rank test* to compare the survival probabilities of several groups. The null hypothesis for this test is that the survival probabilities for all groups are the same.

2.3.2 Cox Proportional Hazard

The cox proportional hazard model is a statistical measure to calculate the instantaneous risk of a subject to experience the event at a given point in time, i.e. the hazard ratio. The formula for this model is as follows:

$$h(t, X) = h0(t) \times \mathrm{e}^{\sum_{i=1}^{p} \beta_i \times X_i} \tag{2.4}$$

$h(t, X)$ describes the hazard at time t under the influence of the variables X. $h0(t)$ is the baseline hazard at time t without the influence of any variables, and the expression at the end is an exponential expression over the sums of the p variables times an regression coefficient. The p denotes the number of variables for which the influence on the survival probability is analyzed.

These hazard ratios can then be interpreted as the probability of a subject to encounter the event,. [DGK12]

2.4 Related Work

The contribution system of open-source software (OSS) projects is a widely researched field in software engineering. In this section, we present research that is related to this thesis. This includes work about the success of contributions to OSS projects, the influence of social factors on reviews, and studies that use survival analyses in the context of OSS projects. The mostly related work, of course, is the study by Bosu et al. [BC14], which we replicate in this study. We present their approach and their results in Section 3.1.

Weißgerber et al. [WND08] investigate how the size of a patch influences the success and the review interval of a patch. They find, that smaller patches have a higher success rate than larger patches but that the size does not seem to influence the review interval of a patch. Another research related to the success of contributions to OSS projects is the study by Jiang et al. [JAG13]. They investigate how to influence the success of a patch on the Linux kernel. They find that the maturity of a patch and the experience of a developer influence the success rate of a submission and that the choice of reviewer, among other things, influences the review interval of a contribution. The third study, we present on the topic of the success of contributions to OSS projects is the study by Rigby et al. [RGS08]. They conduct their research on the Apache HTTP Server project and are able to find that the majority of contributions are done by the core group of developers. Moreover, they find that the overall review interval for patches in this project is very short and conclude that this means that most of the contributions are very small.

Since we calculate the coreness-score of a developer in an OSS project using network metrics, we present a study by Joblin et al. [JAHM17] on this topic. They compare

count-based and network-based metrics for the use of classifying developers into core and peripheral groups. Although, we do not sort the developers of OSS projects into core and peripheral groups, we use the same metrics to calculate the coreness-score. Count-based scores are the commit count and the number of mails a developer sends, to name a few. Types of network-based metrics are the eigenvector centrality and the hierarchy, which we presented in Section 2.2.2. They find that classifications using network-based metrics are more widely agreed with among the developers of OSS projects but are not better than classifications using count-based metrics.

Another related topic is the influence of social factors on reviews in OSS projects. Bosu et al. [BCB+17] find that there are four general social factors that influence the review of a contribution to an OSS project. Although they analyze projects from *Microsoft* for their study, they state that since the contribution system is the same as for most OSS projects, this should also hold for such projects. They find that a good relationship between the author of a patch and the reviewer can have a positive effect on the review. Moreover, just like in the original study, they find that a good reputation of a developer can have a positive effect on the review of a patch. The third factor they identify to have a positive influence is the area of expertise of a reviewer. If the reviewer has a lot of experience in the field the patch lies in, there is a higher probability that the reviewer will prioritize the review of such a patch. The last factor they identify is that reviewers tend to take on reviews with low anticipated effort. This is consistent with the findings of Weißgerber et al. [WND08], that smaller patches have a higher probability of being accepted.

The last related topic, we present in this part of the thesis are studies that use survival analyses in the context of software engineering. Lin et al. [LRS17] use survival analyses to find what factors influence developer turnover in OSS projects. They find that the probability of developers leaving an OSS project is lower if they start to contribute to the project earlier, mainly modify files instead of creating them, and mainly write code and not documentation. The next study that uses survival analyses in the context of OSS development is a study by Ortega et al. [OI09]. They apply survival analyses to estimate how long a developer stays with an OSS project. They find that the mean survival time of developers in OSS projects lies in the interval of 500 to 1,000 days. The last research we present on this topic is a study by Samoladas et al. [SAS10]. They show how to develop a framework, using survival analyses, that predicts the probability of termination or continuation of an OSS project.

3 Original Study

The goal of the study by Bosu et al. is to investigate whether the reputation of developers has an influence on the success of their contributions to open-source software (OSS) projects. Meaning, they want to find out if a developer with a better reputation has a generally better outcome of the patch-submit-review process, which we introduced in Section 2.1.1. [BC14]

3.1 Hypotheses

To fully be able to present the study by Bosu et al., we have to present their hypotheses. The general research question is concerned with whether the reputation of developers has an influence on their contributions to OSS projects, as we described in the previous section. Since the reputation of an developer is a very complex social structure, they have to find some sort of proxy for this. Bosu et al. choose the core-periphery structure of an OSS project for that. The core-periphery structure is a classification approach for developers in OSS projects. We present their approach to classify developers in Section 3.3.

Once the developers are classified, Bosu et al. identify four main metrics to investigate the success of contributions to OSS projects: the *first-feedback interval*, the *review interval*, the *code-acceptance rate*, and the *number of patch revisions per review request*.

The first-feedback interval is defined as the period between the first submission of a patch and the fist review comment or answer on this submission. They pose that the first-feedback interval of core developers is shorter than the interval of peripheral developers as hypothesis. Moreover, they state this should be because core developers should know which reviewer to choose for their submission and that their prior relationship with the reviewer can lead to the reviewer prioritizing the review.

The review interval is defined as the time that elapses from the first submission of a patch until the end of the review process. Bosu et al. pose the hypothesis that the

review interval is shorter for the submissions of core developers than the submissions of peripheral developers. They state that this should be for similar reasons as the shorter first-feedback interval for core developers.

The code-acceptance rate is defined as the rate of accepted submissions of a developer over the number of all submitted patches of a developer. Bosu et al. state that this rate is higher for core developers than for peripheral developers. The reason for this is that core developers should be more familiar with the code base than peripheral developers and thus should be able to produce higher-quality contributions.

Finally, the number of patch revisions per review request is defined as the number of reworks a developer has to make of his patch until it is accepted. When a reviewer identifies a problem with a contribution, the developer that sent it has to fix the problem and resubmit it. The re-submission is called a *revision* and the set of re-submissions is called the *revision set*. Bosu et al. pose the hypothesis that core developers need less revisions of a patch than peripheral developers.The reasoning behind this hypothesis follows the reasoning for the code-acceptance rate.[BC14]

3.2 Data Extraction and Network Construction

In Section 3.1, we described the hypotheses of the study by Bosu et al. [BC14]. To investigate these hypotheses, they extract data of eight different OSS projects. These projects are Chromium OS, ITK/VTK, LibreOffice, OmapZoom, OpenStack, OVirt, Qt Project, and Typo3. While we use OSS projects that use a mailing-list as contribution system, as we described in Section 2.1.1, Bosu et al. analyze projects that use GERRIT[1]. This is a code-review tool where developers can upload their patches onto the platform. The whole review process is then done on the platform.

To extract these data sets, Bosu et al. build a miner similar to the one by Mukadam et al. [MBR13]. They extract the data for the OSS projects under investigation with this miner.

Subsequently they build developer networks, as we described in Section 2.2, on the basis of interactions. In detail, they build undirected, weighted, and simplified networks, i.e. networks with only one edge between developers that interacted with each other and the edge has the number of interactions as weights.

3.3 Core-Periphery Detection

There are two groups of developers in OSS projects. The core developers and the peripheral developers [CWLH06]. The core developers are the usually smaller group containing the people that do most of the work in OSS projects and thus have a deep knowledge of it. The peripheral developer group contains all other developers.

As we stated before, Bosu et al. use the core-periphery structure as a proxy for reputation, i.e. core developers have a better reputation than peripheral developers. For that reason they develop their own approach to classify developers: the *Core*

[1] https://www.gerritcodereview.com/

Identification using K-means (CIK). This approach requires them to first apply some network metrics to the built network. They use 6 different measures for this. Then they combine these metrics using the K-means clustering algorithm [Mac67].

Table 3.1 shows an excerpt of the extracted and classified data. One can clearly see that although the core developers are a significantly smaller amount of people, they are almost always responsible for the majority of the commits and reviews.

Project	# of dev	# of core dev	# of peripheral dev	commits by the core	reviews by the core
ChromiumOS	642	79	533	64.7%	72.5%
ITK/VTK	244	19	225	57.0%	77.2%
LibreOffice	207	20	187	37.6%	88.0%
OmapZoom	642	34	608	34.3%	60.2%
OpenStack	1880	128	1752	53.6%	66.0%
OVirt	193	20	173	51.3%	61.1%
Qt Project	888	63	825	55.9%	66.1%
Typo3	387	30	357	56.3%	71.0%

Table 3.1: Excerpt of general project data taken from Table 2 in the study by Bosu et al. [BC14]. *Developers* is abbreviated with *dev*.

3.4 Results

In this part, we present the results of the study by Bosu et al. [BC14] for the hypotheses, we presented in Section 3.1.

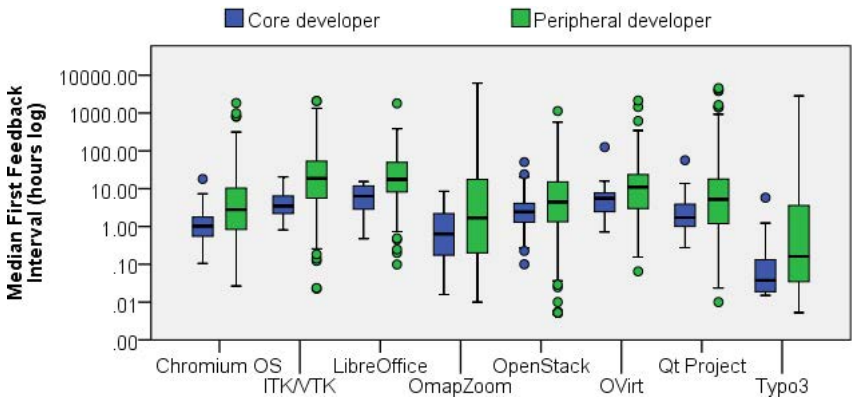


Figure 3.1: Result plot for the first-feedback interval hypothesis of the original study. It shows that core developers have a significantly shorter interval than peripheral developers. This figure is taken from Figure 3 of the original study [BC14].

The results for the first hypothesis, the first-feedback interval, are shown in Figure 3.1. They show that the time that elapses from the first submission of a patch until the first review comment is 1.8 to 6 times lower for core developers than for peripheral developers. For this reason, Bosu et al. accept the first hypothesis.

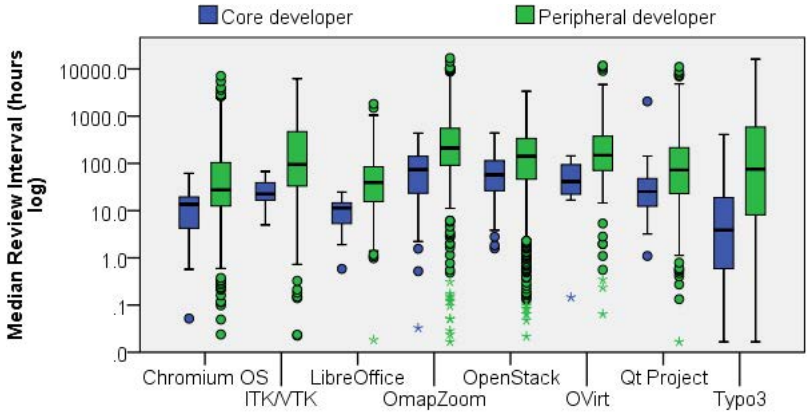


Figure 3.2: Result plot for the review interval hypothesis of the original study. It shows that core developers have a significantly shorter interval than peripheral developers. This figure is taken from Figure 5 of the original study [BC14].

The results for the second hypothesis about the review interval suggest, as shown in Figure 3.2, that core developers have a 2 to 19 times shorter review interval for their contributions than peripheral developers. This supports the second hypothesis and thus, they accept it.

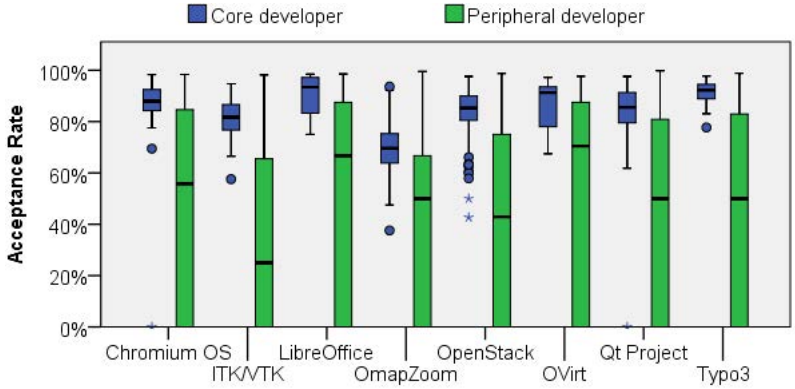


Figure 3.3: Result plot for the code acceptance rate hypothesis of the original study. It shows that core developers have a significantly higher rate than peripheral developers. This figure is taken from Figure 7 of the original study [BC14].

The third hypothesis is concerned with the code-acceptance rate. The results that are presented in Figure 3.3 suggest that the code acceptance is significantly higher for core developers. They accept the third hypothesis.

The fourth hypothesis is about the number of patch revisions per review request. The results are shown in Figure 3.4. The diagram shows that there does not seem to be a significant difference between the number of patch revisions for core developers and the number for peripheral developers. The data even shows that in some cases, core developers need more patch revisions until their contributions finish the review process. For this reason, they state that the hypothesis is overall inconclusive.

Finally, although the fourth hypothesis shows no significant results, they accept the research question. This means that the outcome of their study is that developers

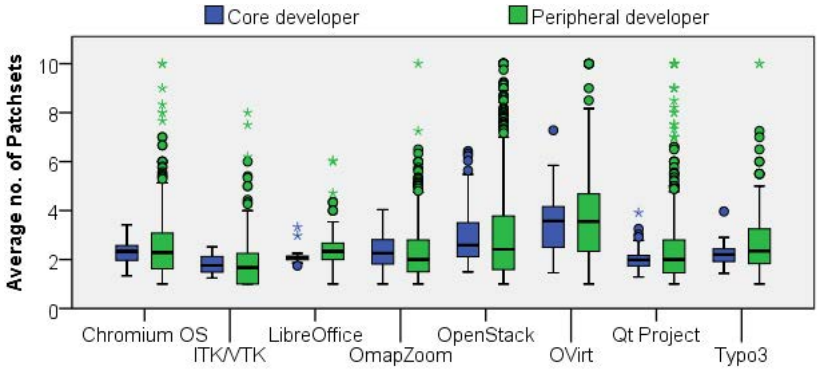


Figure 3.4: Result plot for the number of patch revisions hypothesis of the original study. The results for this hypothesis are overall inconclusive. This figure is taken from Figure 9 of the original study [BC14].

with a higher reputation in an OSS project, indeed seem to have more success with their contributions.

4 Thesis Objective

In this chapter, we present the approach, research question, and hypotheses. First, we pose the research question and hypotheses of this thesis. Subsequently, we present our approach to generate the necessary data for investigating the hypotheses and the methodology, we use to investigate them.

4.1 Hypotheses

In this part of the thesis, we present the research question and the hypotheses of the study. Since we replicate a study by Bosu et al. [BC14], we pose a similar research question as they do: *How does the coreness of a developers influence their contributions to OSS projects?* Contrary to the core-periphery structure that is used in the original study, we use the coreness of developers, as described in Section 2.2.2, as proxy for the reputation of a developer. Our hypotheses also follow the hypotheses of the original study with some slight modifications and one addition. We present the hypotheses in the following.

4.1.1 First-Feedback

Similar to Bosu et al. [BC14], we define the first-feedback interval as the time that elapses from the first submission of a patch until the first answer on the submission. This interval should be shorter for developers with a higher coreness score, because they are usually very good embedded in the project. This means that they should be well known and have good relationships with several reviewers. Moreover, they usually know which reviewer to address for their submission. In contrast to the original study, we investigate not only the interval until a first feedback but also whether there is a feedback on a patch submission at all. Developers with a higher coreness score should, for the same reasons as for the interval, have a higher rate of answered submissions than developers with a lower coreness score. Therefore, we pose the following hypotheses:

Hypothesis H1. *Developers with a higher coreness score have a shorter first-feedback interval than developers with a lower coreness score.*

Hypothesis H2. *Developers with a higher coreness score have a higher feedback rate than developers with a lower coreness score.*

4.1.2 Review Interval

The next objective, we investigate is the review interval. We define this as the time from submission of a patch until the acceptance of the patch. For similar reasons as for the first-feedback interval, developers with a higher coreness score should have a shorter review interval than developers with a lower coreness score. For this reason, we pose the following hypothesis:

Hypothesis H3. *Developers with a higher coreness score have a shorter review interval than developers with a lower coreness score.*

4.1.3 Code Acceptance

The goal of every contribution to an OSS project is its acceptance. This is the ultimate goal of every submission by developers and thus a very important objective to investigate. We state that developers with a higher coreness score should have a higher probability of acceptance for their patches. The reasons for that are that core developers are usually more experienced with the project and have a deep knowledge of the code base. Therefore, they should be able to produce higher-quality code. We pose the following hypothesis concerning the code acceptance:

Hypothesis H4. *Developers with a higher coreness score have a higher code-acceptance probability than developers with a lower coreness score.*

4.1.4 Number of Patch Revisions

The last objective of this thesis is the number of patch revisions a developer needs to submit until the patch is accepted. A patch revision, as we described in Section3.1, is a enhancement of a patch following a review comment. Developers with a higher reputation and thus a deeper knowledge of the OSS project should be able to write higher-quality patches. For this reason, they should need less revisions of a patch until its acceptance. Therefore, we pose the following hypothesis:

Hypothesis H5. *Developers with a higher coreness score need less patch revisions until the patch gets accepted than developers with a lower coreness score.*

4.2 Approach

In the following, we describe the detailed approach including the data extraction, data preparation, and the concrete approach to analyze the hypotheses, we posed in Section 4.1.

4.2.1 Case Studies

To investigate our research question regarding correctness, we have to analyze OSS projects. We analyze nine different case studies that all use a mailing-list as a contribution tool, contrary to the original study [BC14]. We described the concrete review process using a mailing-list in Section 2.1.1. We use such projects, since the data is easily available. The nine case studies we investigate are JAILHOUSE[1], BUSYBOX[2], FFMPEG[3], GCC[4], GIT[5], LLVM[6], QEMU[7], U-BOOT[8], and FLAC[9].

4.2.2 Data Extraction

To analyze the nine case studies, we presented in the previous section, we have to extract the raw data sets of the OSS projects. The three things, we need to extract are a list of all commits, a list of all mails, and a list of all authors of the projects. This is the basis for all our analyses. The data is extracted using CODEFACE[10], a tool developed by *Siemens*, which is able to extract all necessary data [JMA+15]. The data is extracted from multiple sources. The commits and authors are extracted from the version-control system of the projects and the mailing list is extracted from publicly accessible `mbox` archives, downloaded from gmane[11].

CODEFACE saves the extracted data into a MYSQL database, from where we then extract it into `.csv` files using a tool called CODEFACE-EXTRACTION[12].

4.2.3 Network Construction

Just like Bosu et al. [BC14], we use the extracted data to build developer networks, which we introduced in Section 2.2.1. To build these networks, we use a `R` library named CORONET[13]. This library implements all necessary functionality to read the data and build configured networks from the data. For visualization reasons we show an example network, built from the data of JAILHOUSE, in Figure 4.1.

For this thesis, we first build three different types of developer networks: *cochange*, *mail*, and *combined* networks. These three types of networks represent the two main types of developer networks, we introduced in Section 2.2.1. The cochange network is a developer network where the edges between two developers represent that these two developers worked on the same file in the OSS project. The mail network, on the other hand, contains edges that represent a mail interaction between two developers.

[1] https://github.com/siemens/jailhouse
[2] https://busybox.net/
[3] https://www.ffmpeg.org/
[4] https://gcc.gnu.org/
[5] https://git-scm.com/
[6] https://llvm.org/
[7] https://www.qemu.org/
[8] https://www.denx.de/wiki/U-Boot/WebHome
[9] https://xiph.org/flac/
[10] https://siemens.github.io/codeface/#/home
[11] http://www.gmane.org
[12] https://github.com/se-passau/codeface-extraction
[13] https://github.com/se-passau/coronet

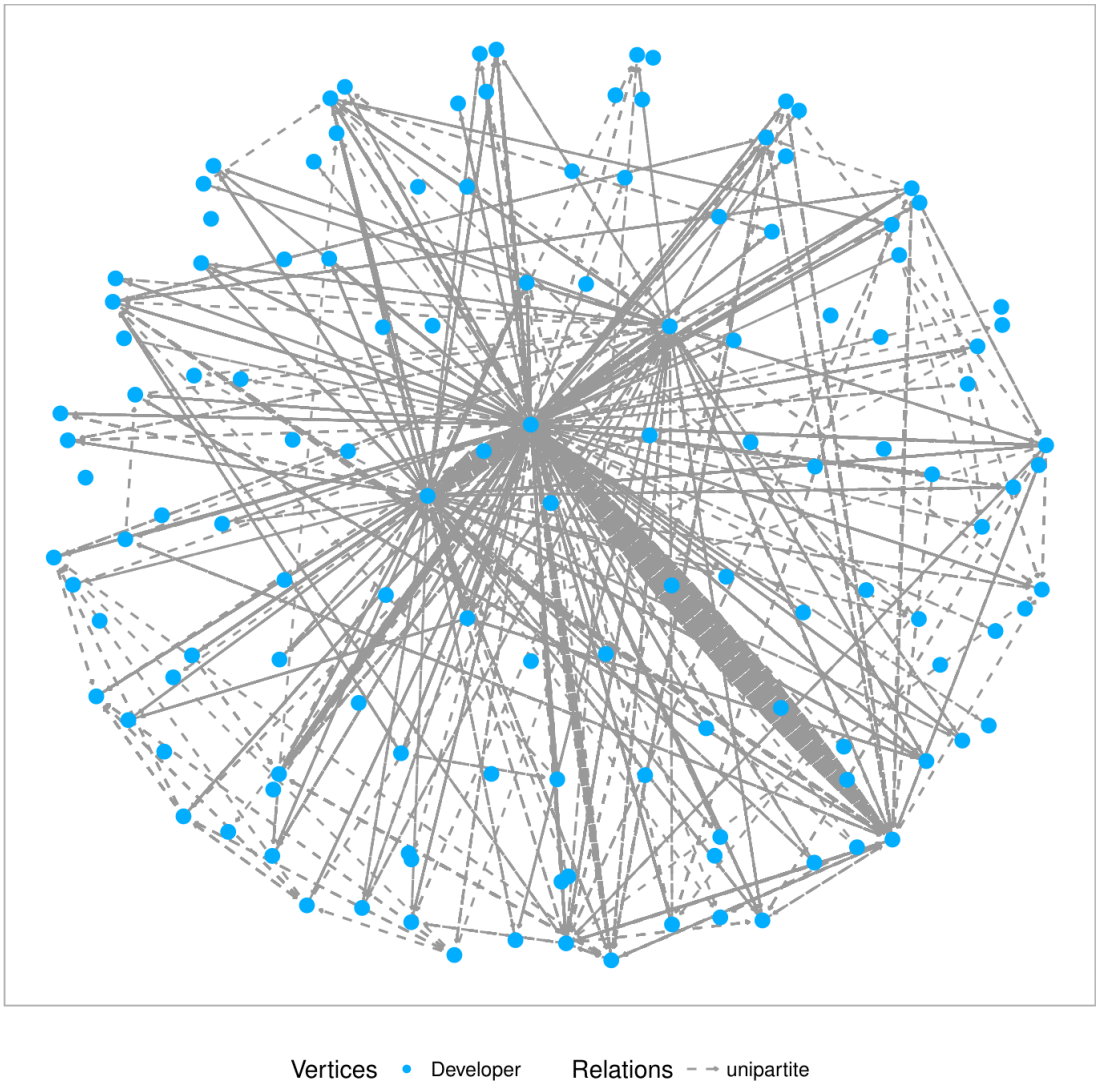


Figure 4.1: Network plot of JAILHOUSE.

The third network includes mail and cochange edges between the developers. We build all of these networks as directed, unweighted networks.

Since we want to reach a higher generalizability with our study than the original study, we split our networks into subsequent ranges of nine months. These splitted networks are then analyzed separately. We do this because there is a high fluctuation in the developers of an OSS project and thus there is some degree of uncertainty whether a classification of developers over the whole life span of a project is robust.

4.2.4 Coreness-Score Calculation

Using the generated networks, we can now calculate the coreness score for all developers in every range they are active in. As we mentioned before, we use the coreness score as a proxy for the reputation of a developer, i.e. the higher the score the higher the reputation.

We define the coreness score as the value of the two network metrics, which we introduced in Section 2.2.2. We calculate these with the help of CORONET and normalize the scores.

Contrary to Bosu et al. [BC14], we do not use these values to classify the developers into the core and peripheral groups. We do this because there is some uncertainty as to how exactly to classify them. Some use the *80/20* method [CWLH06], where all developers with a value over the 80th percentile get classified as core and the rest into the peripheral group, while others use completely different approaches. To avoid this, we decided to use the coreness score without the classification for a more fine-grained approach.

4.2.5 Mapping Patches and Commits

The final step for the analysis of our hypotheses is the mapping of patches and commits. Since we analyze OSS projects that use a mailing list as contribution tool, we do not know which mails contain patches and which are just replies or discussions. Bosu et al. [BC14] are able to extract this data from GERRIT, as well as the information about the commits that followed a successful patch. We have to find a way to detect these things.

We do this using a tool for the analysis of patch stacks called PASTA[14]. The specifics about what a patch stack is and how it is analyzed using PASTA does not concern the topic of this thesis so we do not explain it in detail. But PASTA has an additional feature called the mailbox analysis, which does exactly what we intend to do. [RLM16]

The first step of the mailbox analysis is the detection of mails containing patches. These are then marked as patches. Then it uses a similarity analysis to find patches that are revisions of one and the same patch. Subsequently the analysis maps the patch set to the resulting commit if there is one. We can then use this data in CORONET, to map the mails containing patches and the commits the patches led to.

[14] https://github.com/lfd/PaStA

4.2.6 Approach for the Analysis of the Hypotheses

The last step towards the results of the study is the analysis of the hypotheses introduced in Section 4.1. Therefore, we use the extracted and prepared data, which we described in the previous sections. We implement the analyses in R and present the detailed approach per hypothesis in the following.

For the analysis of the hypotheses H1 through H4, we use survival analyses, as introduced in Section 2.3. Namely, we use the Kaplan-Meier method for the hypotheses H1 and H3, and the cox proportional hazard for the investigation of H2 and H3. To apply a survival analysis to the data, we have to bring the data into a certain format. We present this format in Table 4.1. The first column contains the IDs of the revision-sets of a casestudy. The second, third, and fourth column are the ones relevant for the survival analyses. The *period* column describes the survival time. The *consored* column describes whether the event has happened or not. A 0-value means that the event has not been observed and thus the data-set is censored. The *coreness* column represents the coreness value of the developer responsible for the revision-set. We obtain the coreness value from our into subsequent ranges split networks, as described in Section 4.2.3.. The coreness represents in detail the reputation of the developers responsible for the revision-sets in the range where they first submitted the patch. The last column represents this responsible developer.

id	period	censored	coreness	author

Table 4.1: Data format for the survival analyses of H1 through H4

For this study, we have two different configuration options. The used network type and the used coreness metric. Therefore, we have to build this data-set six times per OSS project. One for every combination of our configurations, e.g., a co-change network with the eigenvector centrality as coreness metric. Therefore, we also calculate six results per case study and hypothesis. The network types are the co-change network, the mail network, and a combined network. The coreness metrics are the eigenvector centrality and the hierarchy.

For the hypotheses H1 and H3, we use the Kaplan-Meier method. Since the log-rank test for the Kaplan-Meier method can only perform the rating when there is a grouped or binary variable of influence, we group the coreness values into two groups. We do this in two different ways. One is to group the data using the 80/20 principle, which is quite common for grouping developers into the core and peripheral groups and was introduced in Section 4.2.4 [CWLH06]. And secondly, we compare the 75th percentile against the 25th percentile. Therefore, we first use R to calculate the survival curves and the log-rank test for the grouped model. We then present the survival curves as Kaplan-Meier plots and interpret the results.

For the hypotheses H2 and H4, we use the cox proportional hazard model. We then present the hazard ratios for every case study and configuration. This is easier to interpret than the Kaplan-Meier model, since we can interpret the results without looking at the survival curves. For illustration purposes, we again group the data and present exemplary plots.

Finally we investigate hypothesis H5. Since the objective of this hypothesis is not suitable for survival analyses, as there is no time parameter, we use a linear regression model to check the hypothesis. Before we do that, we arrange the data into the format shown in Table 4.2.

id	number of revisions	coreness	author

Table 4.2: Data format for the linear regression analysis of H5

The first, second, and fourth column are the same as their respective columns in the data format for the survival analyses. The *number of revisions* column shows how many revisions of the same patch were needed until it was accepted.

Subsequently, we build a linear regression model and present exemplary scatterplots.

For all the statistical tests we use in this thesis, we use the significance level of 5%, i.e. p-values below 0.05 indicate a significant result.

5 Results and Discussion

In this chapter, we present the results of our analyses and discuss them in detail. We only present exemplary plots for the hypotheses so that the chapter does not overflow with plots. The rest of the plots can be found in the appendix of the thesis.

5.1 Descriptive Statistics and Patch Detection

In Section 4, we presented the way to extract the data necessary for our analyses. Here, we present an excerpt of the data extracted. In detail, we present the number of commits, mails, and authors, we obtained per OSS project, the number of patch-sets of the project, and the number of accepted patch-sets.

5.1.1 Descriptive Statistics of our Subject Projects

Project	# commits	# mails	# authors
LLVM	158,519	708,065	6,407
GCC	158,615	574,165	9,636
QEMU	46,578	430,561	7,205
Git	34,872	338,500	9,246
U-Boot	44,680	319,160	7,924
FFmpeg	80,535	242,295	5,998
BusyBox	14,259	42,013	2,736
Jailhouse	1,786	5,619	131
FLAC	3,815	3,880	594

Table 5.1: Number of the extracted commits, mails, and authors for each project.

The extracted data for the OSS projects we analyze differs significantly in terms of size. We include small projects and large projects in this study to see whether there is an influence of the size of a project on the study results. The sizes of the projects can be seen in Table 5.1. These number of extracted data differ slightly

form the number of data actually used in the study, since the time-frames these mails or commits lie in differ slightly and to analyze them correctly, we unionize the time-frames within an OSS project.

5.1.2 Patch Detection and Mapping Results

Project	# patches	# patch-sets	# accepted sets
LLVM	262,537	236,912	3,041
GCC	76,197	59,338	9,873
QEMU	190,308	65,391	35,937
Git	113,560	57,752	24,241
U-Boot	111,561	45,851	32,950
FFmpeg	24,287	14,525	7,466
BusyBox	2,136	1,595	335
Jailhouse	3,210	1,438	1,177
FLAC	414	268	121

Table 5.2: Number of detected patches, patch-sets, and successful patch-sets for each project.

As we described in Section 4.2.5, we use PaStA to detect patches, group them into patch-sets, and map these to commits. The results of this analysis are shown in Table 5.2. Notable is that we detect a very low number of patches for some OSS projects, like BusyBox. This could mean that the developing community does not use the mailing-list extensively. The number of accepted patch-sets on the other hand is very low in relation to the number of patch-sets for some projects, like LLVM or GCC. This could either mean that there are very few contributions that are accepted after the review or that the detection has some flaws.

5.2 Results

In this section, we present the detailed results for the hypotheses we introduced in Section 4.1. We present exemplary plots for each of the hypotheses. Finally, we decide whether we accept, reject, or have to conclude that the hypothesis is inconclusive.

5.2.1 Hypothesis H1: First-Feedback Interval

The first results we present are the results for the hypothesis regarding the first-feedback interval. As stated in Section 4.1.1, we hypothesize that the first-feedback interval is shorter for developers with a higher coreness score and thus a higher reputation within the OSS project. Since we are not able to calculate the Kaplan-Meier curves with a log-rank test for the coreness-scores individually, as described in Section 4.2.6, we group the values into two groups and analyze these.

The results show mixed indications. There are configurations present - a configuration consisting of project, network type, and classification method - that indicate the approval of the hypothesis. But there are also results that show developers with

a lower coreness-score have a higher probability to get a first feedback over time and even results that show no difference at all.

The plots include the following: The diagram on top shows the Survival probability over time for the two different groups. The legend above shows the colors of the survival curve per group. For the 80/20 grouping, the label *CL=<80%* represents all developers with a coreness score below the 80% quantile and the label *CL=>80%* respresents the group of developers with a coreness-score of over the 80% quantile. Analogously, the label *CL=25th* represents the group of developers with a coreness-score in the 25th percentile and the label *CL=75th* the group of developers with a coreness-score within the 75th percentile for the comparison of the two percentiles. The table on the bottom shows how many developers are still "alive" per group and point in time. The event or "death event" for this analysis is the reception of a first feedback on a submission. So a lower survival probability indicates a shorter first feedback interval.

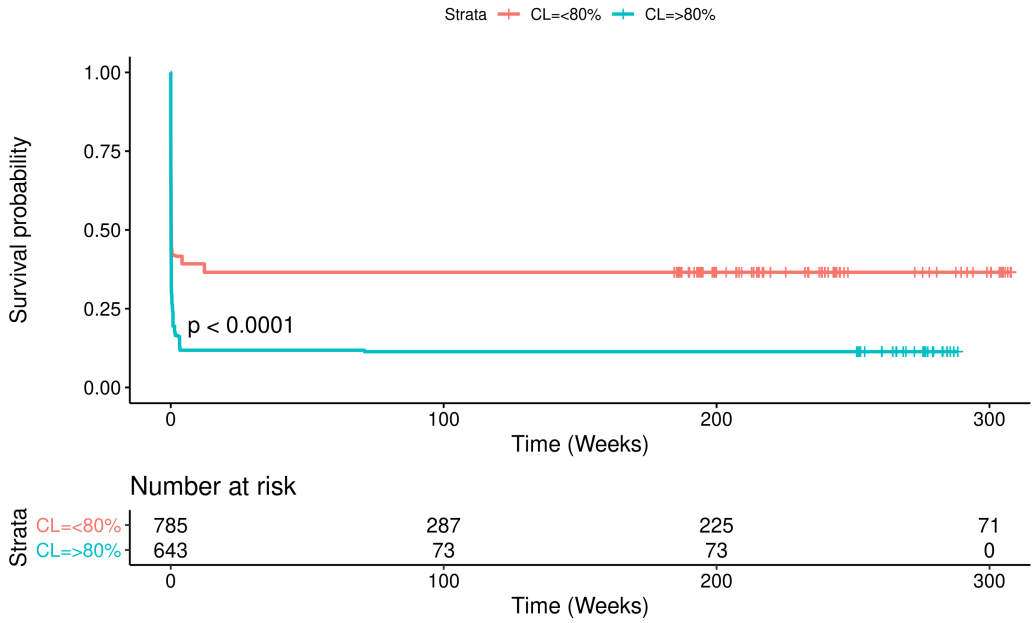


Figure 5.1: The Survival curves for the first-feedback interval of JAILHOUSE analyzed on the mail-network with an eigenvector-centrality classification with the 80/20 grouping.

The results for JAILHOUSE with an eigenvector classification on a mail-network, as seen in Figure 5.1, tend towards the acceptance of the hypothesis. The p-value for the log-rank test shows a significant difference between the curves of the two groups. Moreover, the curve for the group with coreness-scores above the 80% quantile shows a lower survival probability and thus a higher probability to receive a faster first feedback over time.

The results for FFMPEG with a eigenvector-centrality classification on the combined network, as shown in Figure 5.2, on the other hand, show the exact opposite. Here the developers with coreness-scores within the 75th percentile seem to have a signif-

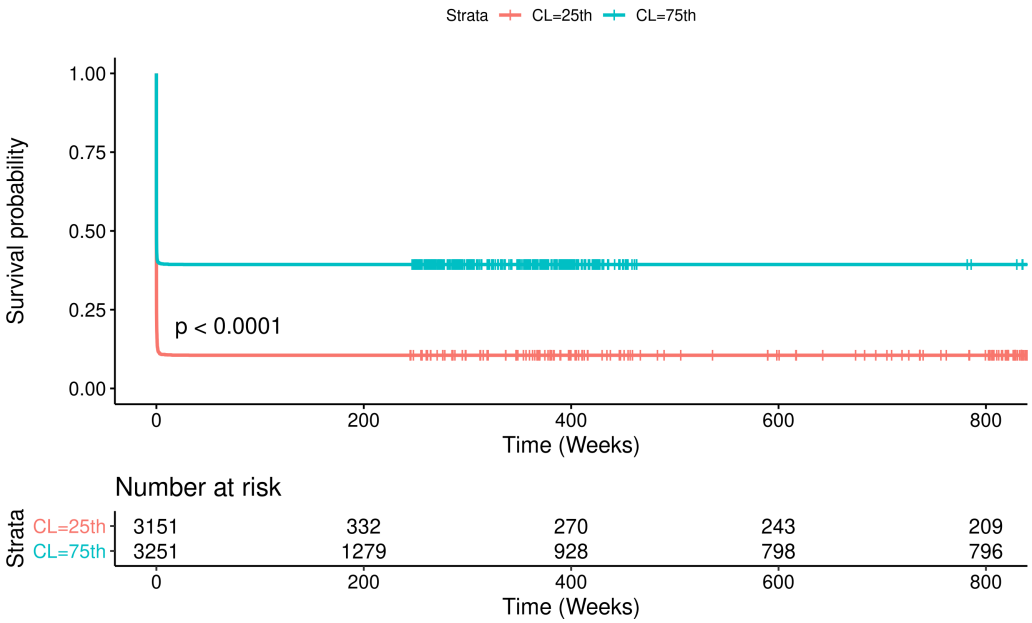


Figure 5.2: The Survival curves for the first-feedback interval of FFmpeg analyzed on the combined-network with an eigenvector-centrality classification and the grouping in 25th and 75th percentile.

icantly lower survival probability over time than the developer with coreness-scores within the 75th percentile. This tends towards the rejection of the hypothesis.

The rest of the plots for all other case studies with all other configurations are shown in the Appendix. To summarize them: These results show a similar picture as the two plots we present here. Sometimes the probability to receive a first-feedback is higher for developer with a higher coreness-score and sometimes the exact opposite seems to be the case. In some cases the p-value of the log-rank test indicates no significant results at all, meaning that there is no observable difference between the two groups.

In conclusion, we deem the results for hypothesis H1 to be **inconclusive**, as there is data that supports either side of the hypothesis and even data showing no statistically significant results at all.

5.2.2 Hypothesis H2: Feedback Rate

The second hypothesis, we posed is that developers with a higher coreness-score have a higher feedback rate, so a higher probability of receiving a feedback at all. For this analysis, we used the cox-proportional hazard model, as described in Section 2.3.2. This model gives us a hazard rate per OSS project and configuration. A hazard rate value over 1 shows that the probability of receiving a first feedback increases with an increasing coreness-score and a value below 1 shows the opposite of that. Moreover, the cox-proportional hazard model gives us a significance value.

The detailed results for the analysis of H2 are presented in Table 5.3. Here we see, that although having a lot of significant results, the hazard rates are not consistent.

This indicates that the hypothesis is neither acceptable nor rejectable. The values for GIT and U-BOOT are the only results consistent with the hypothesis, as they indicate over all configurations that the higher the coreness-score of a developer, the higher the probability to get a feedback on a submission. Other OSS projects show inconsistent results, like GCC, where the hazard rate changes from above 1 to below one depending on the configuration. And then there are case studies, like FFMPEG or JAILHOUSE that indicate that developers with a lower coreness-score have a higher probability of receiving a feedback on their submissions.

As described in Section 4.2.6, we then grouped the data again to compare groups of coreness-scores against each other. These results are then plotted and we present some of these plots in the following:

The labels for the forest plots are the same as for the survival-curve diagram, which we explained in the previous section. The upper white part of the diagram shows the reference group. In our case this is always the group of developers with the lower coreness-scores. The lower gray part shows the hazard ration for the other group of developers in relation to the reference group. The N below the labeled of the groups shows the number of events (feedback receptions) there were in the group of developers. The value next to the label of the gray group shows the hazard ratio itself.

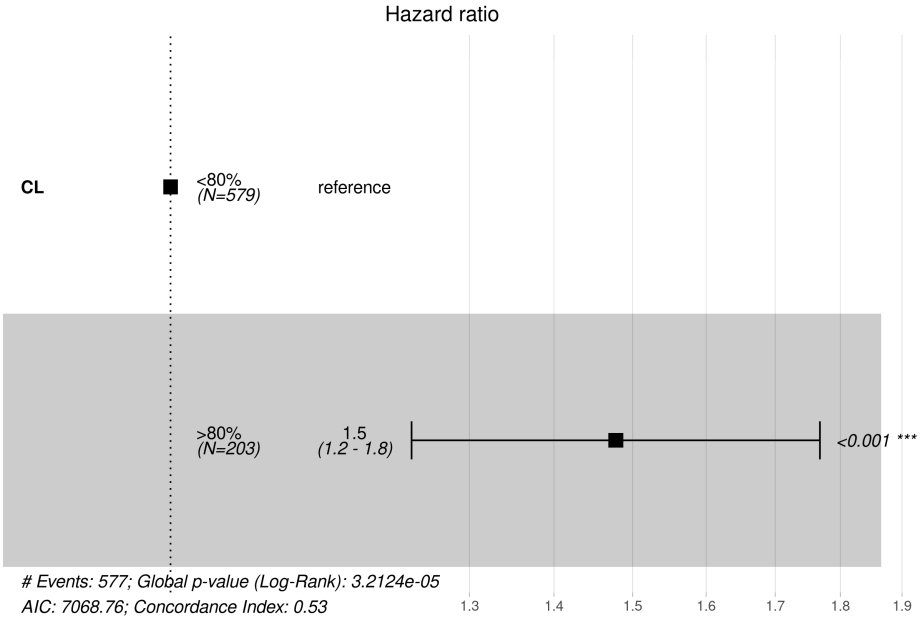


Figure 5.3: The forest plot for the Hazard ratio of the first feedback of GCC analyzed on the co-change network with an eigenvector classification and the grouping with the 80/20 method.

In Figure 5.3, we present the results for GCC with an eigenvector classification on the co-change network. This plot shows the same result as the result for this configuration in Table 5.3. The hazard ratio and thus the probability to receive a feedback on a submission is higher for developers that have a higher coreness-score than for developers with a lower coreness-score. This supports the hypothesis.

Project	**Eigen cochange**	**Eigen mail**	**Eigen combined**	**Hier cochange**	**Hier mail**	**Hier combined**
LLVM	0.11***	1.37***	1.08	0.44***	0.44***	0.13***
GCC	13.73***	1.04*	12.21***	7.39***	0.83***	35.98***
QEMU	1.59***	1.02	1.88***	1.16***	1.01	1.37***
Git	1.31***	1.26***	1.33***	1.15***	1.06***	1.14***
U-Boot	1.26***	1.23***	1.23***	1.25***	1.25***	1.23***
FFmpeg	0.34***	1.04	0.52***	0.40***	0.58***	0.52***
BusyBox	1.16	1.18	1.18	0.75*	0.87	0.73
Jailhouse	0.38***	0.87	0.55***	0.29***	0.42***	0.32***
FLAC	0.67	1.59	0.38	8.18*	2.99	2.18

Table 5.3: Hazard rates for the feedback rate. Values higher than 1 show a higher probability to get a feedback for developer with a higher coreness-score. *: p-value $<$ 0.05, **: p-value $<$ 0.01, ***: p-value $<$ 0.001. eigenvector centrality is abbreviated with Eigen and hierarchy is abbreviated with Hier.

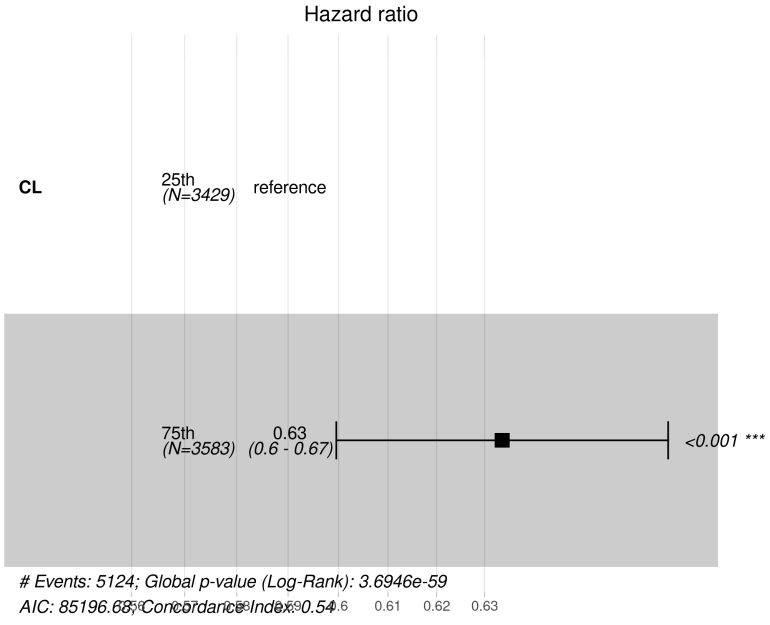


Figure 5.4: The forest plot for the Hazard ratio of the first feedback of FFmpeg analyzed on the co-change network with an eigenvector-centrality classification and the grouping in 25th and 75th percentile.

The plot shown in Figure 5.4 on the other hand shows that for FFmpeg with an eigenvector classification on the combined network the hazard rate is lower for developers with a higher coreness-score. This supports the results for this configuration in Table 5.3.

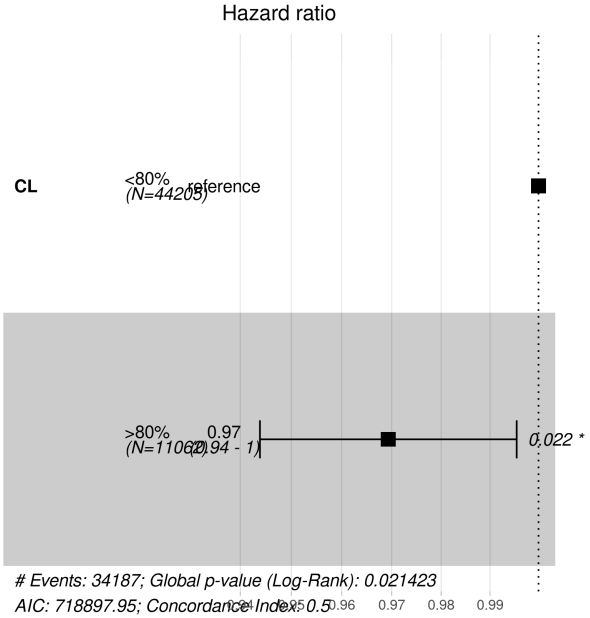


Figure 5.5: The forest plot for the Hazard ratio of the first feedback of GCC analyzed on the combined network with a hierarchy classification and the grouping with the 80/20 method.

But the grouping of the developers by coreness-scores also brings some uncertainty into the model. When we look at the result for GCC with a hierarchy classification on the combined network in Table 5.3, we see that developers with a higher coreness-score should have a higher probability of receiving a feedback on their submissions. But the plot for the grouped developers in Figure 5.5 shows the exact opposite of this. So we see, that the results for the un-grouped data sometimes show a difference to the results for the grouped data. We discuss this in detail in Section 5.3.2.

Since there is data that supports the positive and the negative side of the hypothesis, as shown in this section, we conclude the results for H2 to be **inconclusive**.

5.2.3 Hypothesis H3: Review Interval

For the analysis of the hypothesis that developers with a higher coreness-score have a smaller review interval, we have the same limitations regarding the log-rank test for the Kaplan-Meier estimator as for the analysis of H1. Therefore, we group the data again using the 80/20 method and the 25/75 method. The results again show mixed indications including the acceptance of the hypothesis, the rejection of the hypothesis, and even results that show no difference for the groups.

This means that there are results that show that developers with a higher coreness-score have a shorter review interval for their submissions, results that show that developers with a higher coreness-score have a longer review interval for their submissions, and that there is no difference in the results for the two groups at all for some configurations.

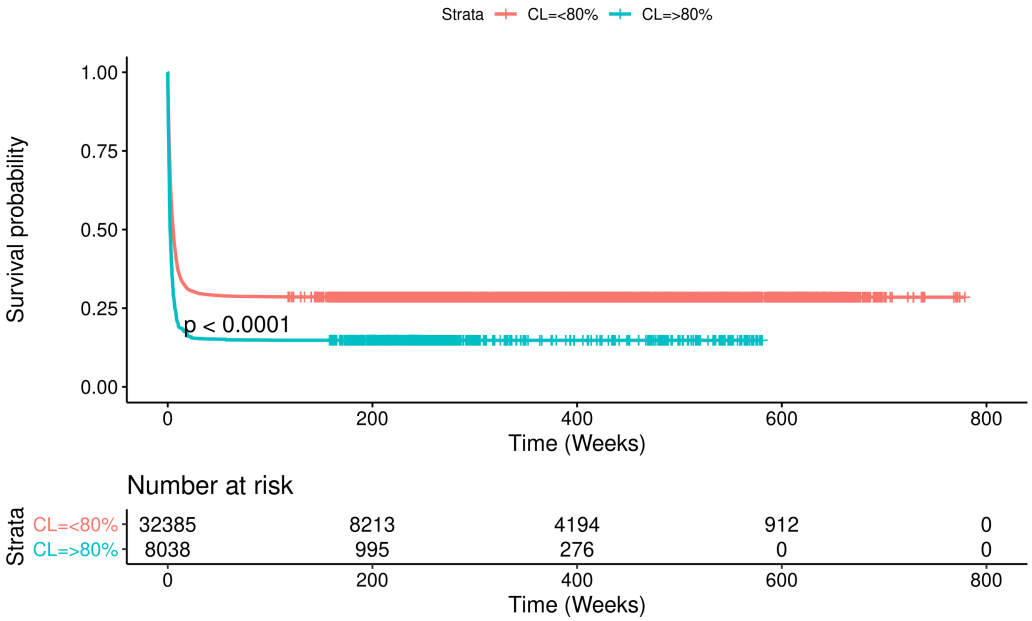


Figure 5.6: The Survival curves for the review interval of U-Boot analyzed on the co-change network with an eigenvector-centrality classification and the grouping with the 80/20 method.

The results for U-Boot with an eigenvector-centrality classification on the co-change network for example, show that the survival probability, i.e., the probability not to finish the review phase successfully is lower for developer with a higher coreness-score. The plot for this is shown in Figure 5.6. This indicates that the hypothesis is true for said configuration. As shown in the risk table on the bottom of the plot, there even is no open submission for developers with a higher coreness-score after 600 weeks, while it takes approximately 800 weeks in maximum for developers with a lower coreness-score.

The results for BusyBox with an eigenvector-centrality classification on the mail network, on the other hand, show that developers with a lower coreness-score have a higher probability to successfully end the review process earlier than developer with

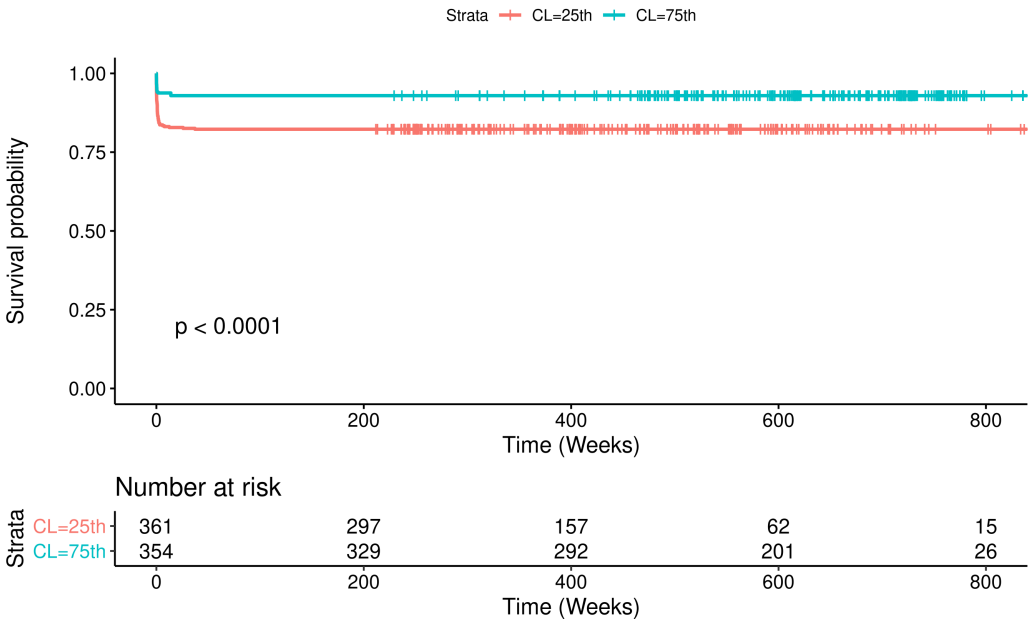


Figure 5.7: The Survival curves for the review interval of BusyBox analyzed on the mail network with an eigenvector-centrality classification and the grouping in 25th and 75th percentile.

a higher coreness-score. This does not support our hypothesis. And the risk table on the bottom shows that there are even some patch-sets that are still not accepted after 800 weeks for both groups.

The rest of the results can be found in the appendix of the thesis. These results show a similar overall picture as the two results we present here. Sometimes, the results indicate that the hypothesis is false, sometimes not. And some of the results show no difference for the groups at all.

Therefore, since we have results that show different results, sometimes supporting the hypothesis and sometimes not supporting it, we conclude the results for the hypothesis overall to be **inconclusive**.

5.2.4 Hypothesis H4: Code-Acceptance Rate

The fourth hypothesis centers on the code-acceptance rate of developers in OSS projects. We hypothesize that developers with a higher coreness-score have a higher code-acceptance rate, i.e., a higher possibility to having their contributions accepted, than developers with a lower coreness-score. This hypothesis is analyzed by using the cox-proportional hazard model. This is analogue to the analysis of H2.

The results per OSS project and configuration are shown in Table 5.4. Just like the results for H2 that we present in Section 5.2.2, there are mixed results present. The hypothesis seems to be true for the OSS projects U-Boot and QEMU, but there are other case studies, like LLVM, where this does not hold. the results for FLAC indicate that there is no difference in the probability to have a submission accepted for developers with higher coreness-score or developers with a lower coreness-score.

Subsequently we present some exemplary plots, again grouping the data using the 80/20 method and the grouping in coreness-scores in the 25th and 75th percentile.

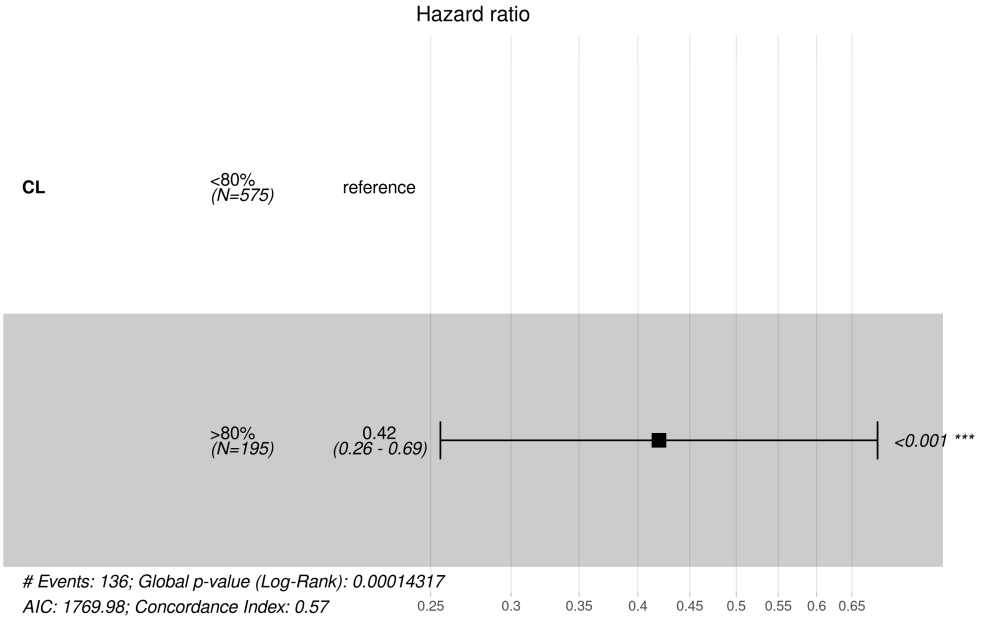


Figure 5.8: The forest plot for the Hazard ratio of the code-acceptance feedback of GCC analyzed on the co-change network with an eigenvector classification and the grouping with the 80/20 method.

The plot shown in Figure 5.8, represents the result for GCC with an eigenvector classification on the co-change network. This result shows, like the result for this configuration without grouping in Table 5.4, that the probability to have a submission accepted is higher for developers with a higher coreness-score than for developers with a lower coreness-score.

The plot shown in Figure 5.9, on the other hand, shows the exact opposite for LLVM with an eigenvector classification on the combined network. Here we see that developers with a lower coreness-score seem to have a higher probability of having their submissions accepted than developers with a higher coreness-score.

As we can see in Table 5.4, there are results that support either side of the hypothesis, just like for the previous three hypotheses. And then there are again results that show no difference in the probability of having a submission accepted for developers with a higher coreness-score versus developers with a lower coreness-score.

In conclusion, we have to deem the results for this hypothesis to be **inconclusive**, since there are a lot of results that do not support the hypothesis but also some that do.

Project	**Eigen cochange**	**Eigen mail**	**Eigen combined**	**Hier cochange**	**Hier mail**	**Hier combined**
LLVM	<0.001***	0.005***	<0.001***	<0.001***	0.75***	1.27
GCC	27.96***	1.37***	11.68***	0.007***	1.94***	32.23***
QEMU	2.76***	1.34***	2.71***	1.78***	1.16***	1.98***
Git	1.63***	1.15***	1.18***	1.36***	0.99	1.03
U-Boot	1.90***	1.43***	1.47***	1.28***	1.43***	1.41***
FFmpeg	0.70***	1.89***	0.96	0.67***	1.05	0.91**
BusyBox	0.62	0.16***	0.89	0.58*	0.09***	0.95
Jailhouse	2.31***	0.81	0.93	3.71***	2.23***	2.91***
FLAC	0.54	1.04	0.95	0,51	0.16	0.17

Table 5.4: Hazard rates for the code-acceptance rate. Values higher than 1 show a higher probability to have the submission accepted for developers with a higher coreness-score. *: p-value < 0.05, **: p-value < 0.01, ***: p-value < 0.001

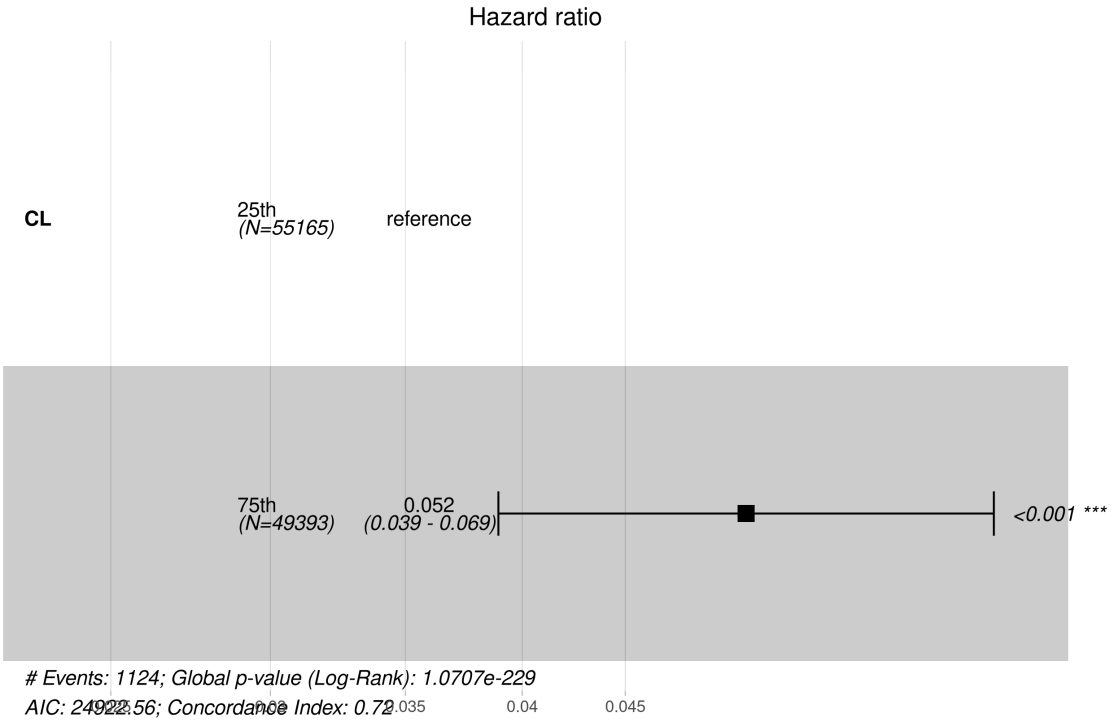


Figure 5.9: The forest plot for the Hazard ratio of the code-acceptance feedback of LLVM analyzed on the combined network with an eigenvector classification and the grouping in 25th and 75th percentile.

5.2.5 Hypothesis H5: Number of Revisions

The last hypothesis for this thesis is centered on the number of patch revisions, a developer needs to send until a patch is accepted. We hypothesize that a developer with a higher coreness-score needs to send less revisions of a patch than a developer with a lower coreness-score. This is the only hypothesis of this thesis we analyze with a linear regression instead of a survival analysis. We do this, since for a survival analysis there has to be a time variable and for this hypothesis we do not analyze some time-specific measure.

Furthermore, we can plot the results of this analysis without grouping the coreness-scores first. To plot them we use scatter plots with the number of revisions on the y-axis and the coreness on the x-axis. Every point on the plot represents a patch-set that needed y revisions and is assigned to a developer with a coreness-score of x. The red line represents the regression line, i.e., whether the number of revisions needed increases with increasing coreness-score or decreases with increasing coreness-score.

Project	**Eigen cochange**	**Eigen mail**	**Eigen combined**	**Hier cochange**	**Hier mail**	**Hier combined**
LLVM	Yes	Yes	Yes	Yes	No	Yes
GCC	No	Yes	No	No	Yes	Yes
QEMU	Yes	Yes	Yes	Yes	Yes	Yes
Git	Yes	No	No	Yes	Yes	Yes
U-Boot	Yes	Yes	Yes	Yes	Yes	No
FFmpeg	Yes	Yes	Yes	Yes	Yes	Yes
BusyBox	No	No	No	No	No	No
Jailhouse	Yes	Yes	Yes	Yes	No	Yes
FLAC	No	No	No	No	No	No

Table 5.5: This table shows whether there is a significant influence of the coreness-score on the number of revisions a developer has to send. *Yes* means there is an influence present and *No* means that there is no significant influence.

The results, whether or not there is a statistically significant influence of a developers' coreness-score on the number of patch revisions are shown in Table 5.5. A Yes-entry means that there is a significant influence, while a No-entry means that there is no significant influence of the coreness-score on the number of revisions.

But the fact that a result is significant or not is not enough to accept or reject our hypothesis. Next, we have to examine the scatter plots of the results to see whether the number of revisions does decrease with increasing coreness-score of a developer, which would support the hypothesis. Therefore, we examine some exemplary plots in this section. The rest of the plots can be found in the appendix.

The plot in Figure 5.10 shows the results for LLVM with an eigenvector-centrality classification on the mail network. Here we can see that the regression line indicates that the number of revisions decreases for developers with a higher coreness-score. This can also be seen by looking at the *Slope* parameter on top of the plot. A negative number here indicates a falling regression line. So these results support the

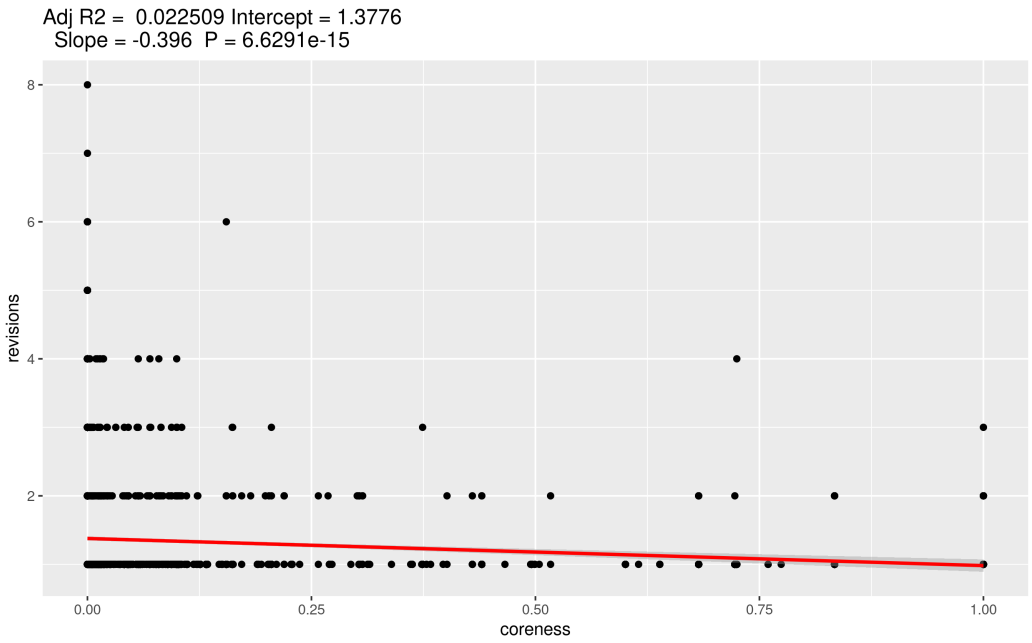


Figure 5.10: The scatter plot for the number of revisions under the influence of a developers' coreness-score of LLVM analyzed on the mail network with an eigenvector classification.

hypothesis. But one other important measure is the R^2 value of the model. This can be found as the *Adj R2* value on top of the plot. This is very low for the analysis of this configuration. But the R^2 value is low for all configurations of our analysis, which brings some uncertainty whether the linear regression is the right model for the analysis of this hypothesis.

In contrast to the result of LLVM, there are also results like the one presented in Figure 5.11. This plot represents the analysis results for JAILHOUSE analyzed on the mail network with an eigenvector classification. Here, we see that the regression line is rising, i.e., the number of revisions increases with an increasing coreness-score of the responsible developer. This supports the opposite of our hypothesis.

Moreover, there are again some statistically insignificant results indicating no influence of a developers' coreness-score on the number of patch revisions at all.

Since there are results of the analysis of this hypothesis that support the hypothesis and also results that do not support the hypothesis, we have to conclude that the results and thus the hypothesis is overall **inconclusive**.

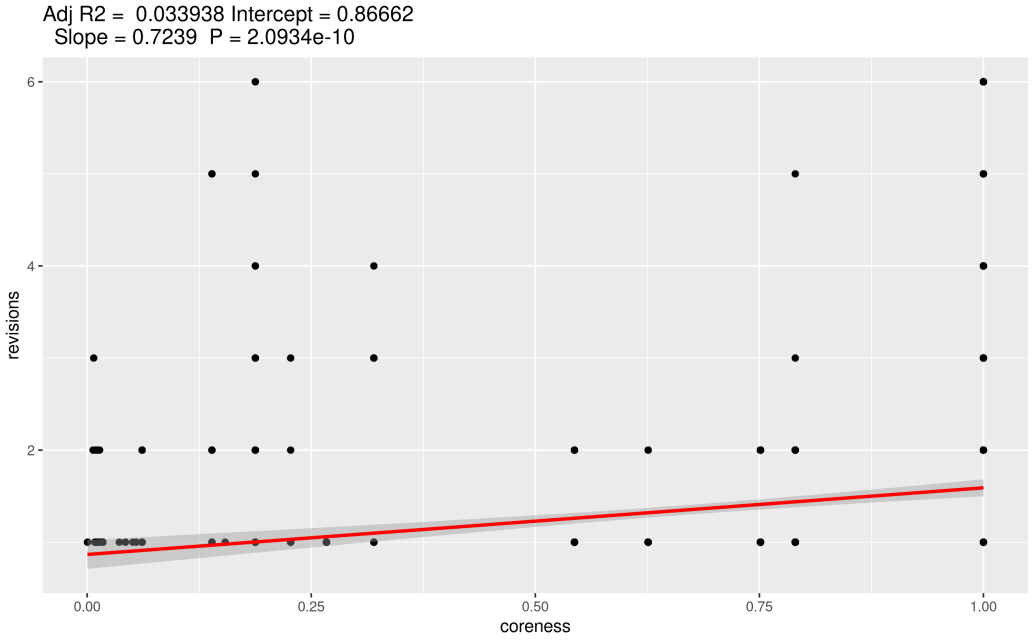


Figure 5.11: The scatter plot for the number of revisions under the influence of a developers' coreness-score of JAILHOUSE analyzed on the mail network with an eigenvector classification.

5.3 Discussion

In this section, we discuss the results, we presented in Section 5.2. Moreover, we discuss points of interest regarding the analyses and chosen OSS projects.

5.3.1 Hypothesis H1: First-Feedback Interval

In the first hypothesis following the first hypothesis of the original study [BC14], we state that developers with a higher coreness-score have a lower first-feedback interval than developers with a lower coreness-score. Bosu et al. are able to accept this hypothesis while we have to deem the results to be inconclusive.

The results, we presented show that the results seem to differ from OSS project to OSS project and even from configuration to configuration sometimes. This indicates that there might be a dependency of the first-feedback interval on a multitude of factors, not only the coreness score of a developer. For example, there might be the directive in the community that contributions of new contributors should be prioritized to ensure their further commitment to the project. This could explain, why we observe for some projects and configurations that developers with a lower coreness-score have a shorter first-feedback interval. Or maybe the interval until there is a first feedback on a submission is dependent on the importance of the submission. So to say an important patch to handle a critical security vulnerability might be reviewed faster than a uncritical documentation change. There might also be a lot more influencing factors on this.

5.3.2 Hypothesis H2: Feedback Rate

The second hypothesis of this thesis is that developers with a higher coreness-score have a higher feedback rate, i.e. a higher probability to get a feedback on a contri-

bution at all, than developers with a lower coreness-score. This hypothesis is not present in the original study by Bosu et al. [BC14] so we can not say whether this holds or not for their study.

Our results show that there is no conclusive answer of the hypothesis possible. Some results indicate that the hypothesis might be true, while others indicate that the exact opposite might be the case. Some results show no tendency towards or against the hypothesis at all.

There might be a multitude of reasons why this is the case. The simplest reason is that there might be a lot of factors that influence whether there is a feedback on a contribution or not. This follows our assumptions on the results of H1. The example of a patch containing a critical security fix versus a patch containing documentation enhancements holds here as well. Moreover, the direct relationship of the author of the patch to the reviewer could be a influencing factor, e.g. a reviewer that knows the author personally could be inclined to giving a feedback on the contribution even if it is an irrelevant patch that could be ignored.

5.3.3 Hypothesis H3: Review Interval

The third hypothesis of our study again follows the original study by Bosu et al. [BC14]. This hypothesis centers on the review interval of a submission, i.e. the time from submission of a patch until the acceptance of said patch. We hypothesize that this interval is shorter for developers with a higher coreness-score than for developers with a lower coreness-score. The results, like the results for H1 and H2, prove to be inconclusive. Bosu et al. are able to accept the hypothesis based on their study results.

The results show that the tendency towards or against the hypothesis differs among the investigated OSS projects and even within the projects depending on the analysis configuration. Some results, on the other hand, show no tendency at all. The main reason for these inconsistencies in the results could either mean that the relationship between the coreness-score of developers and the review interval of their submissions might be purely random and not a general observation for OSS projects. So to say, it might be present in some OSS projects while the length of the review interval depends on different factors in other OSS projects.

Another influencing factor could again be the importance of the patch as described in the previous two results discussions. For example a patch containing a critical enhancement could be pushed through the review process faster than a simple documentation or maintainability enhancement.

5.3.4 Hypothesis H4: Code-Acceptance Rate

The fourth hypothesis of our thesis deals with the code-acceptance rate of developers, i.e. the rate of accepted versus not accepted patches a developer submits to an OSS project. We hypothesize, following the original study by Bosu et al. [BC14], that this rate is higher for developers with a higher coreness-score than for developers with a lower coreness-score. Bosu et al. are able to accept this hypothesis while we have to deem the results to be inconclusive.

In detail, the results we presented in Section 5.2.4 show that for some OSS projects and study configuration there is an indication towards the acceptance of the hypothesis, while other results suggest the opposite. Some results again show no tendency towards a higher code-acceptance rate of developers with a higher coreness-score or towards a higher code-acceptance rate for developers with a lower coreness-score at all. This indicates that the coreness-score is not a robust measure to predict the code-acceptance rate of developers but rather the code-acceptance rate depends on a combination of the coreness-score with other factors or might even be completely independent.

One other influencing factor could be the quality of a submission. Although we hypothesized that developers with a higher coreness-score should be able to produce higher-quality patches because of their previous involvement and usually deep knowledge of the code base, there could also be high-quality contributions by new developers.

5.3.5 Hypothesis H5: Number of Revisions

The last hypothesis of this thesis, following the original study by Bosu et al. [BC14], is that developers with a higher coreness-score need less revisions until a submission is accepted. Bosu et al. conclude that the results of their study for this hypothesis are inconclusive but tend towards the acceptance of the hypothesis. We also conclude that the results of our study are inconclusive but we are not able to discern a tendency towards the acceptance or rejection of the hypothesis.

The results indicate that there might be an influence of the coreness-score of developers on the number of patch revisions they need to send until acceptance. But the results are not consistent on whether the number of revisions increases or decreases with an increasing coreness-score of a developer. This seems to differ from OSS project to OSS project and even within an OSS project depending on the network type and classification metric. This could either mean that the number of revisions is independent from the coreness-score of a developer and the results are completely random or that there are other factors that influence the number of revisions developers need until their contributions are accepted.

One such influencing factor could be that developers with more experience in an OSS project might be inclined to take on more difficult tasks and thus produce more complex contributions that need more review and enhancements until the acceptance than simpler changes. This would lead to developers with a higher coreness-score needing more revisions of a patch until acceptance.

5.3.6 Differences to the Original Study

In this thesis, we replicate a study on the same topic by Bosu et al. [BC14]. Since we use a different approach and analyze different OSS projects than the original study, we have to emphasize the differences in detail.

Firstly, Bosu et al. analyze OSS projects that use Gerrit as contribution tool. We analyze only OSS projects that use a mailing-list for this purpose. Moreover, Bosu et al. build interaction networks for their case studies, while we build three

different types of developer networks: mail networks, co-change networks, and a combination of the two. This adds to the probability of not being able to reproduce the results of the original study. Furthermore, the researchers of the original study classify the developers of the OSS projects they analyze into core and peripheral groups, while we mostly only use the coreness-score of a developer as a proxy for the reputation of a developer. Additional, we use different network metrics to calculate the coreness-score of a developer than Bosu et al. use to classify the developers into groups.

The hypotheses of our study and the original study are mostly the same. We adapt the hypotheses of Bosu et al. to fit our approach and add one more hypothesis. The added hypothesis is that developers with a higher coreness-score have a higher feedback rate on their contributions than developers with a lower coreness-score. We add this hypothesis since it is an interesting research objective on this topic. For the analysis of the hypotheses, Bosu et al. calculate the mean values of the first-feedback interval, the review interval, the code-acceptance rate, and the number of revisions for the core and the peripheral developer groups. Subsequently, they compare these mean values and decide whether or not they accept their hypotheses on the basis of these mean values. We, on the other hand, use the data of the OSS projects we analyze for survival analyses to find the probability of an event to happen. The events in our case are the reception of a feedback and the acceptance of the submission. We then interpret these probabilities to decide if we accept the hypotheses or not. For the hypothesis about the number of revisions a developer needs to send until a contribution is accepted, we use a linear regression model.

5.3.7 Differences to our Previous Study

Since we have already replicated the study by Bosu et al. [BC14] in a previous Bachelor Thesis, we have to discuss the differences in the outcomes and the approach of the previous study and this thesis. In the previous study, we used only three OSS projects and only built the mail network. Moreover, we followed the classification approach of Bosu et al. and sorted the developers into the core and peripheral groups for the analysis. This sorting brings some uncertainty, because there is no unified approach to do this and thus there is no widely approved consensus of what a core and a peripheral developer are in terms of their coreness-score. In this study we try to avoid this uncertainty by not grouping the developers for most of our analyses. Furthermore, we did not split the networks into ranges for the previous study. The problem with analyzing the un-split network is that there might be wrong data as to who is core and who is peripheral. For example a developer that contributes a lot to an OSS project for one year should be considered core in this time-frame but is overall classified as peripheral since the one year is not weighted as much when looking at the whole lifespan of a project. We try to avoid such a scenario in this thesis by splitting the networks into subsequent nine-month ranges.

As in this thesis, we were not able to reproduce the results of the original study in the previous study.

5.3.8 Characteristics of the OSS Projects

In this part, we discuss the characteristics of the OSS projects we analyze, which we presented in Section 4.2.1. The first notable difference between the projects is the

size. The two biggest OSS projects are LLVM ant GCC. Compared to the smallest projects, which are Jailhouse and FLAC, these projects have about 150 to 50 times larger in numbers of commits and even larger in numbers of mails. This gives us a good range for our analyses as we include a variety of very large projects until as very small projects in the study.

The next notable thing about the different projects are the results for the mapping of patches and commits. For example when looking at the numbers for LLVM in Table 5.2, we see that only a very small amount of patch-sets are mapped to commits. This could either mean that there is a mailing-list containing patches we did not analyze or that the mailing-list is not used by the majority of contributors that have accepted contributions. When looking at the number of overall patches for the project, it could also be that we are not able to map a lot of these patches to commits although they were accepted.

When looking at the number of mails for BusyBox in Table 5.1 and then looking at the number of detected patches in Table 5.2, we see that there is only a very low percentage of patches among the mails. This could again mean that there is an additional mailing-list for patches, we did not find or that there is a lot of development going on in the project that is not communicated on the mailing-list.

These uncertainties in the data could of course lead to corrupted results, which we discuss in the threats to validity of this study in Chapter 6.

5.3.9 The Difference Between Linear Regression and Correlation

For the analysis of the fifth hypothesis of our study, as described in Section 4.2.6, we use a linear regression. When using this thype of analysis, one has to discuss the difference between linear regression and correlation.

Correlation describes an analysis that tries to find a relationship between two variables: and independent x-variable and a dependent y-variable. In our case the independent variable is the coreness-score of a developer and the dependent variable is the number of revisions a developer needs to send until a patch is accepted.

Linear regression on the other hand, is a type of analysis that estimates the value of the dependent variable by looking at the independent variable. So when using the linear regression, we can not fully say that there is a correlation between the two. And as we can see from the plots for the linear regression in Section 5.2.5, we see that there are a lot of values that are far away from the regression line and thus the estimated value. So when only looking at the results of the linear regression model, we can not fully decide whether or not there is a correlation between the coreness-score of a developer and the number of revisions the developer needs to send until the patch is accepted.

6 Threats to Validity

Subsequent to the presentation and discussion of the study results, we discuss the threats to validity of our study in this chapter.

One threat is the selection of OSS projects for the analysis. We only analyze projects that use a mailing-list as contribution tool and thus can not generalize the findings of this study to all OSS projects. Since there are a lot of different types of contribution tools for OSS projects, we only cover a small amount of such projects with our analyses. Moreover, we can not guarantee that the analyzed mailing-lists are the only ones containing patches for the analyzed OSS projects. We selected the main development lists for all of the case studies of this thesis but there might be more mailing-lists on which the development is done. So we can not guarantee that we analyze the complete data.

Bosu et al. [BC14] use data mined from Gerrit, as described in Section 3.2. They are able to mine all data necessary for their analyses from this platform. We, on the other hand, have to rely on the correctness of the tool we use to extract and prepare the data. First of all, we have to rely on Codeface to extract all necessary data for our study from the OSS projects since this data forms the basis of all our analyses. The next tool we have to rely on is PaStA to detect all patches on the mailing-lists of the projects and to map all patches that led to commits to their respective commit. There is some uncertainty regarding this, as the analysis method of PaStA is a similarity analysis on the text of a patch on the mailing-list. This type of analysis always brings some uncertainty as to the correctness of the data. As we can see in the data of LLVM in Table 5.2, there is a very low mapping rate of patches to commits for some OSS projects. This could be due to some undetectable mappings.

The next threat to validity is the detection of the coreness-scores of developers. We calculate these scores with two different network metrics but we are not sure whether these metric values are a good proxy for the coreness-score of developers in the analyzed OSS projects.

One other threat to validity is that there might be an influence of the dependent variables of our study on the independent ones. For example, if a developer has to submit a lot of revisions for one patch, the coreness-score increases for the classification using the mail network. This is due to the fact that every submitted revision has to be sent in a mail and thus the developer has more mails and gets a higher coreness-score in our analysis.

The next threat to validity is the fact that we can only analyze the hypotheses H1 and H3 with the Kaplan-Meier method if we group the coreness-scores of developers. This grouping can lead to corrupted results, because there is no generalized method to group the coreness-score of developers and we have to rely on the chosen methods to be correct forms of grouping.

Moreover, we obtain very low goodness-of-fit (R^2) values for the linear regression analysis of H5. This could mean that although the results are significant in most cases, there is no correlation between the coreness-score of developers and the number of revisions they have to submit until a patch gets accepted. This could be dependent on other factors like the complexity of a patch or the code-quality of a patch. So we can not generalize the findings of the analysis of this hypothesis.

Finally, we assume that the coreness-score of a developer is a proxy for the developers' reputation within a project and that the reputation has an influence on the outcomes of review requests a developers sends to an OSS project. This might not be the case. There could be a lot of influencing factors we do not consider in this study. And these influencing factors could also differ from OSS project to OSS project. This would then lead to our study not being generalizable for all OSS projects.

7 Conclusion

In this chapter of the thesis we summarize the findings of our study. Moreover, we give an outlook of further work we will conduct on this topic.

7.1 Summary

In this thesis we investigated whether there is an influence of a developers reputation on the outcomes of the review f contributions a developer makes to an OSS project. This study is a replication of the work by Bosu et al. [BC14], for the purpose of confirming or refusing their results. The detailed study objectives surrounding the review of contributions a developer makes to OSS projects are the time until a developer gets a first feedback on a contribution, the probability to get a feedback at all, the time until the acceptance of a contribution, the probability to get their contributions accepted at all, and the number of revisions a developer has to send until a contribution is accepted.

To analyze these study objectives, we extracted data of nine different OSS projects that all use a mailing-list as a contribution tool. These nine projects are: Jailhouse, BusyBox, FFmpeg, GCC, Git, LLVM, QEMU, U-Boot, and FLAC. Then, we used the coreness-score of developers as proxy for their reputation. For the calculation of the coreness-scores of developers, we built three different types of developer networks: mail networks, co-change networks, and a combination of the previous two. Subsequently, we split the networks into nine-month ranges. We then applied two different network metrics separately to the split networks to obtain the coreness-score for the developers of an OSS project. The two network metrics are the eigenvector centrality and the hierarchy. For the analysis of the hypotheses, we then applied different regression analyses: The Kaplan-Meier method, the cox-proportional hazard model, and linear regression.

Unlike Bosu et al., we were not able to accept the hypothesis that developers with a higher reputation have a shorter first-feedback interval than developers with a lower reputation within an OSS project. We found that this seems to differ between the analyzed projects. While it seems to be true for some projects, we were not able to

confirm it for the majority of projects. Furthermore, we were not able to confirm that developers with a higher reputation are more likely to receive a feedback at all on their contributions. We were again able to confirm this in some cases while the majority of the results shows a tendency towards the rejection of the hypothesis. Therefore, we had to conclude that the results are inconclusive. The next objective was the review interval of submissions to OSS projects. Unlike the original study, we were not able to show that this interval is shorter for developers with a higher reputation within an OSS project. Moreover, we were not able to confirm the results of Bosu et al. regarding the code-acceptance rate. While the original study shows that developers with a higher reputation have a higher code-acceptance rate, we can not confirm this. Our results indicate that this might be dependent on other or more factors than just the reputation of a developer. The last objective of this thesis and the original study, was the number of revisions developers need to send until a patch is accepted. Just like Bosu et al., we were not able to show that developers with a higher reputation within an OSS project need less revisions.

In conclusion, we could not confirm the findings of the original study by Bosu et al. The reputation of developers within an OSS project does not seem to have a generally significant influence on the outcomes of review requests. Therefore, we can only conclude that the reputation of developers can not be the only factor that influences the outcomes of the review process within OSS projects.

7.2 Future Work

Since we will conduct further research on this topic by us in the future, we give an outlook on this here.

The first thing we should do is including more OSS projects into the study. This would make our research more generalizable over all OSS projects. But to reach a higher generalization we should also include OSS projects with different contribution systems. One of the most famous OSS project, we will analyze is the Linux kernel. This is one of the largest and most used OSS project in the world.

Furthermore, we should make the coreness-score metric more robust. Currently, we only use the coreness-score of the developer that is responsible for the initial patch of a revision-set within the range this patch is submitted in. But we should also consider the coreness-scores of the developers that contribute to the revision-set by sending a revision. This could be the same developer that sent the initial patch but there could also be other developers contributing. Moreover, we should consider the coreness-score not only for the range a developer sends the initial patch but all coreness-scores of developers that send a revision in the respective range. For this reason, we could calculate an active range for every revision-set, i.e., from the initial submission until the acceptance of the revision-set and use the coreness-scores of all involved developers in this range.

To decrease the problem of the coreness-score of a developer for a patch that is submitted on the border of the range, we should also introduce a sliding-window approach for the splitting of the networks or calculate the range individually for each revision-set, as we described before.

Next, we have to find a solution for the analysis of the number of revisions a developer has to submit until the acceptance of a patch. The linear regression model does not seem to be a good model for this as is evident with the low R^2 values for the results of this analysis. Moreover, we have to find a solution regarding the Kaplan-Meier method. As we described in the thesis, the log-rank test of this survival analysis is not able to handle continuous variables, which the coreness-score is. We have to find a possibility to analyze the hypotheses, we use the Kaplan-Meier method for, without grouping the coreness-scores.

A Appendix

In the appendix, we present additional plots for the analysis results. Since there are too many plots to to display, we only present chosen ones.

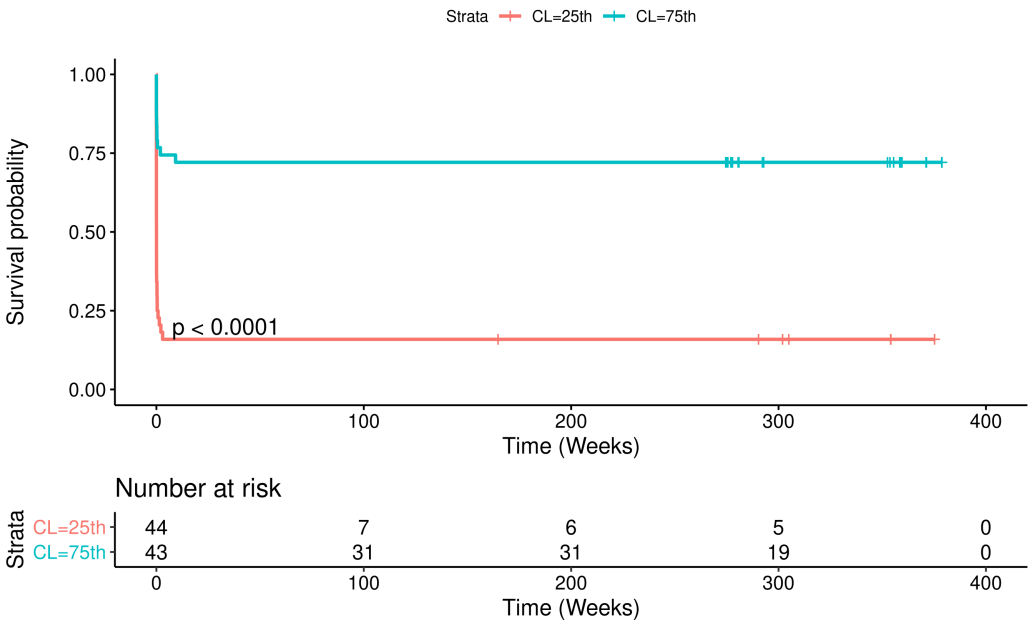


Figure A.1: The Survival curves for the first-feedback interval of FLAC analyzed on the combined-network with an eigenvector-centrality classification and the grouping in 25th and 75th percentile.

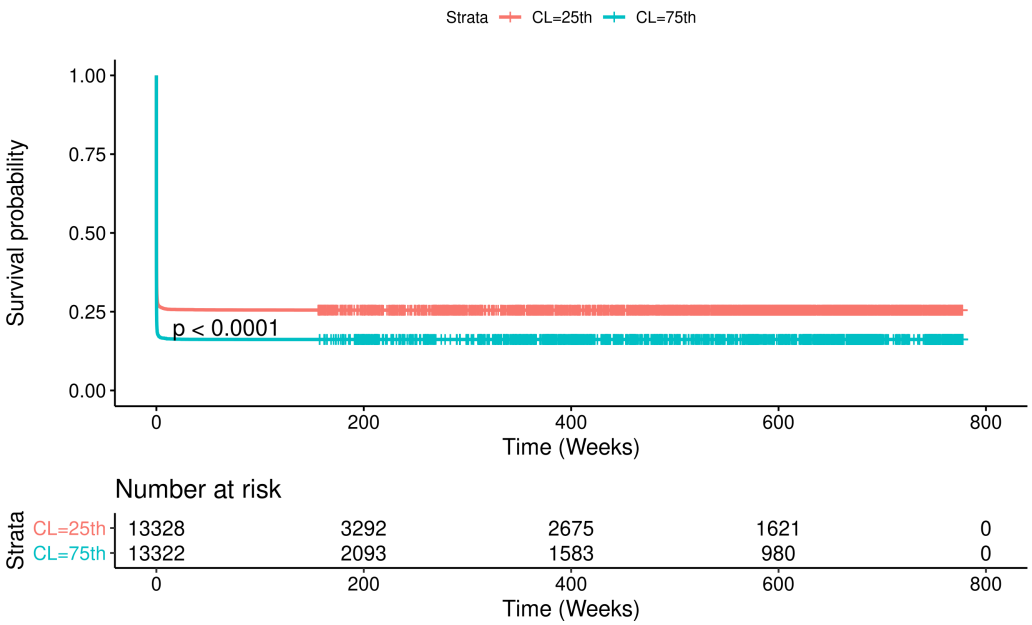


Figure A.2: The Survival curves for the first-feedback interval of GIT analyzed on the combined-network with an eigenvector-centrality classification and the grouping in 25th and 75th percentile.

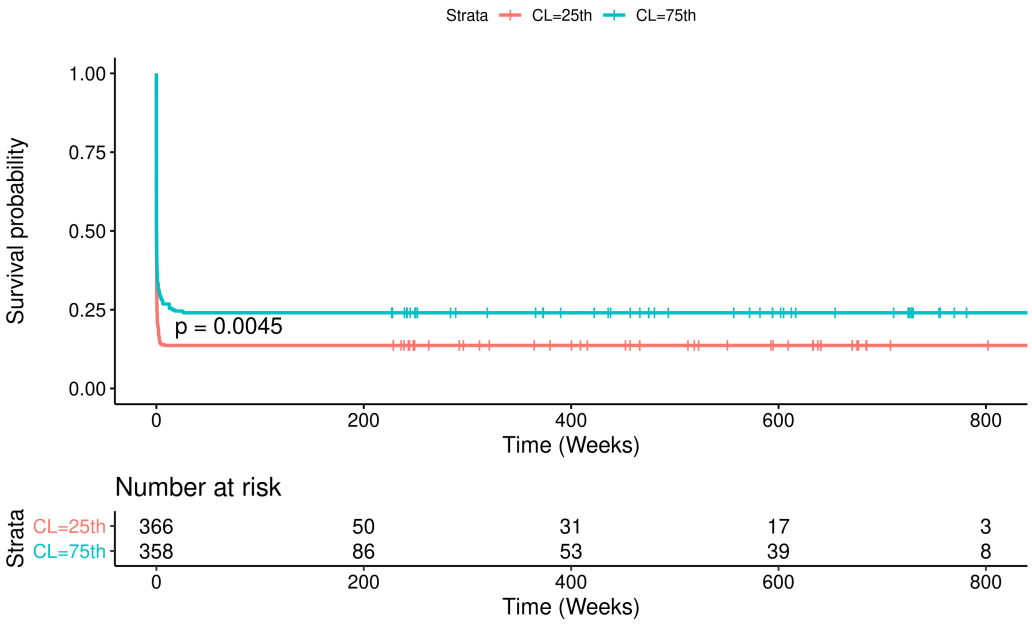


Figure A.3: The Survival curves for the first-feedback interval of BUSYBOX analyzed on the combined-network with an hierarchy classification and the grouping in 25th and 75th percentile.

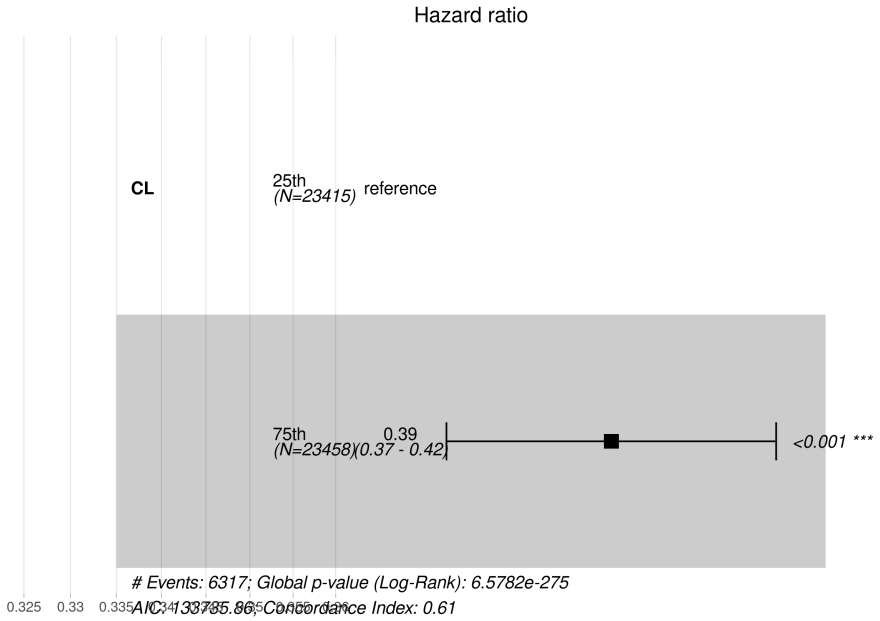


Figure A.4: The forest plot for the Hazard ratio of the first feedback of LLVM analyzed on the co-change network with an eigenvector-centrality classification and the grouping in 25th and 75th percentile.

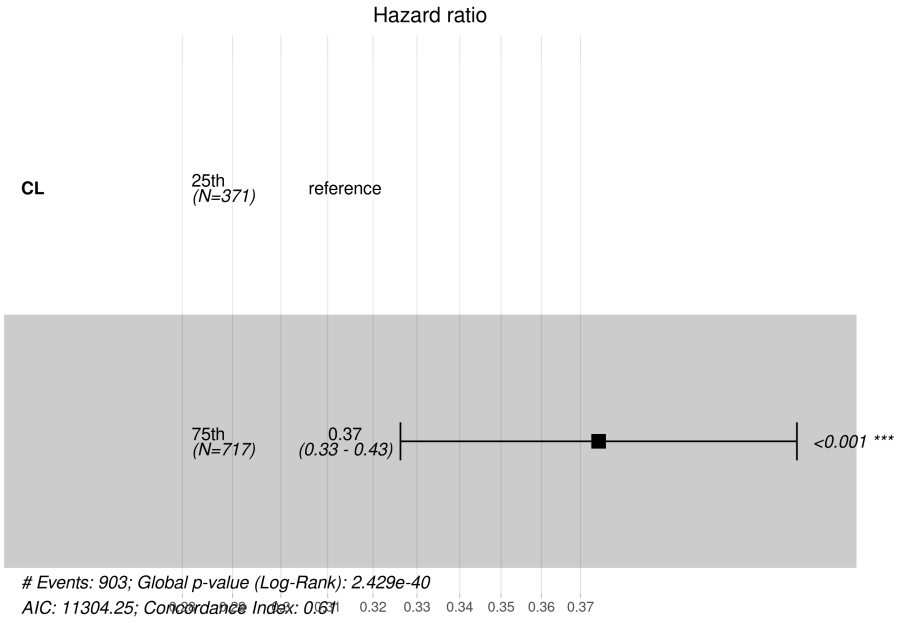


Figure A.5: The forest plot for the Hazard ratio of the first feedback of Jailhouse analyzed on the co-change network with an eigenvector-centrality classification and the grouping in 25th and 75th percentile.

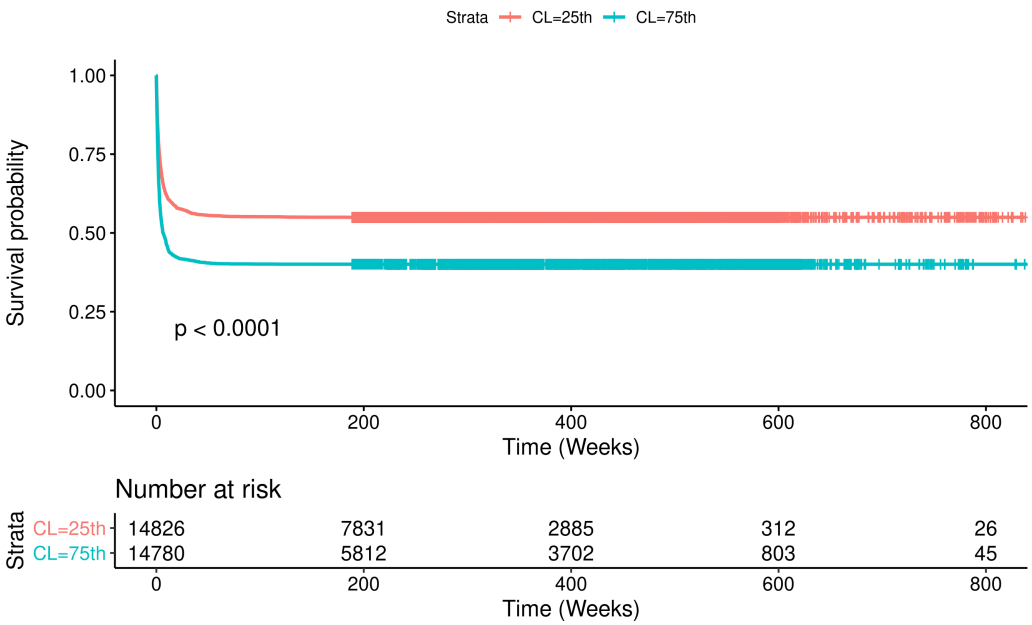


Figure A.6: The Survival curves for the review interval of QEMU analyzed on the mail network with an eigenvector-centrality classification and the grouping in 25th and 75th percentile.

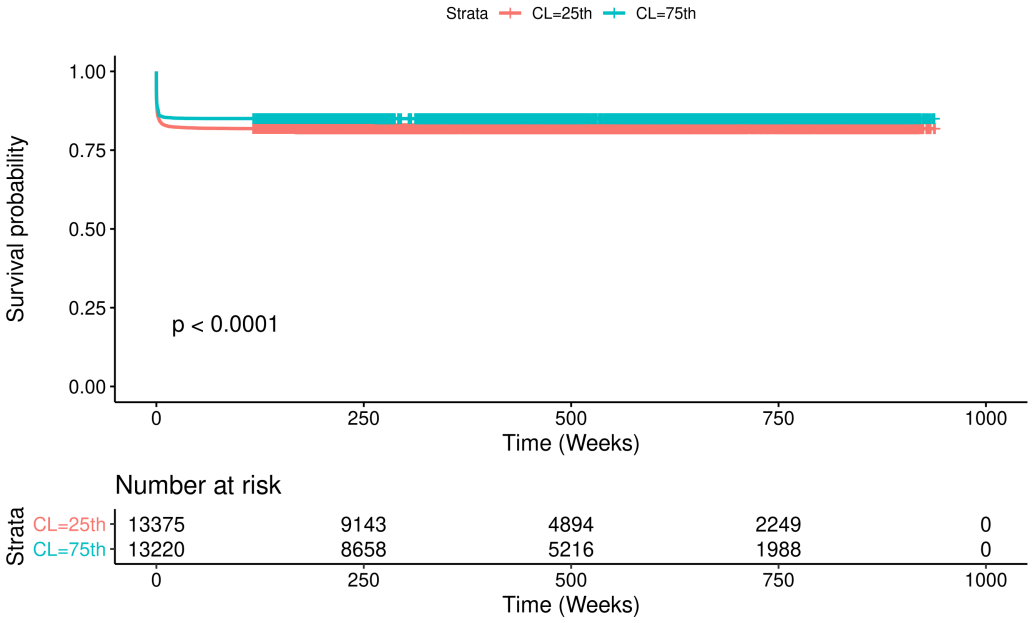


Figure A.7: The Survival curves for the review interval of GCC analyzed on the mail network with an hierarchy classification and the grouping in 25th and 75th percentile.

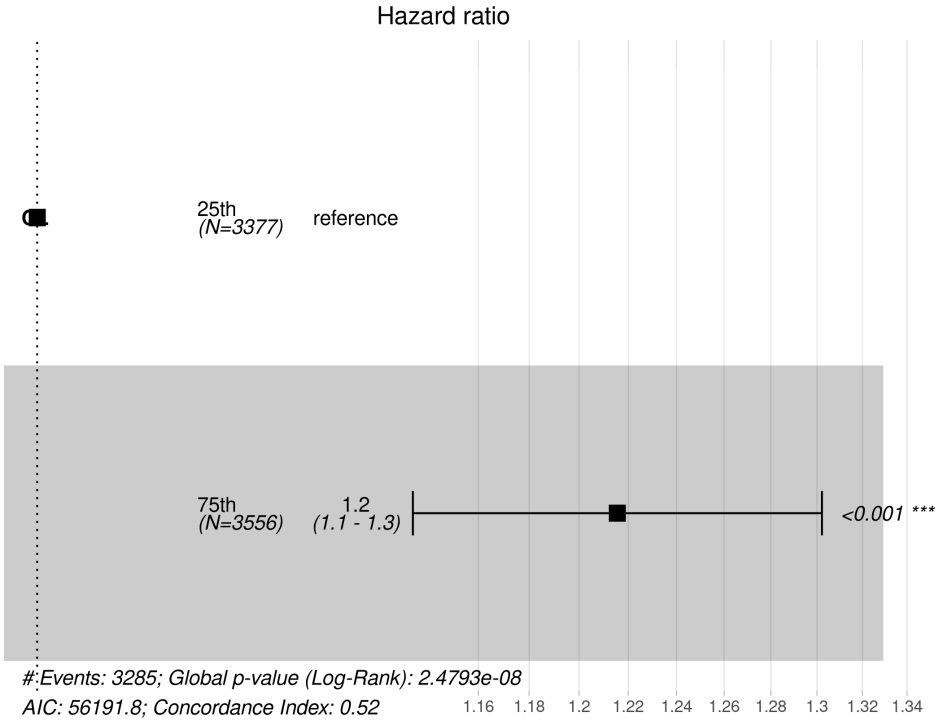


Figure A.8: The forest plot for the Hazard ratio of the code-acceptance feedback of FFmpeg analyzed on the combined network with an eigenvector classification and the grouping in 25th and 75th percentile.

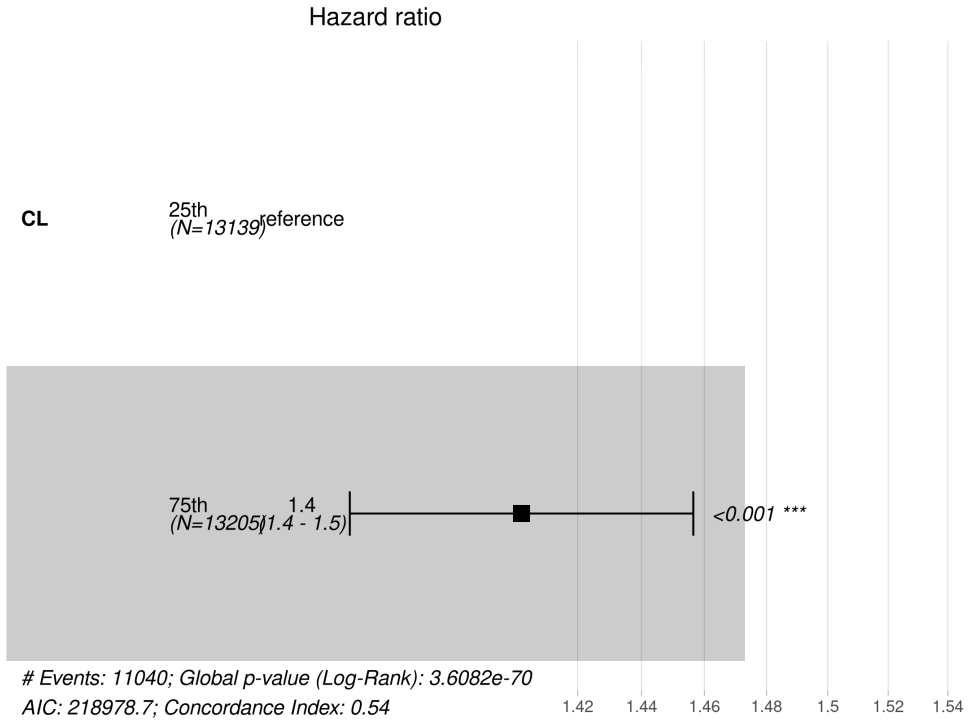


Figure A.9: The forest plot for the Hazard ratio of the code-acceptance feedback of Git analyzed on the combined network with an eigenvector classification and the grouping in 25th and 75th percentile.

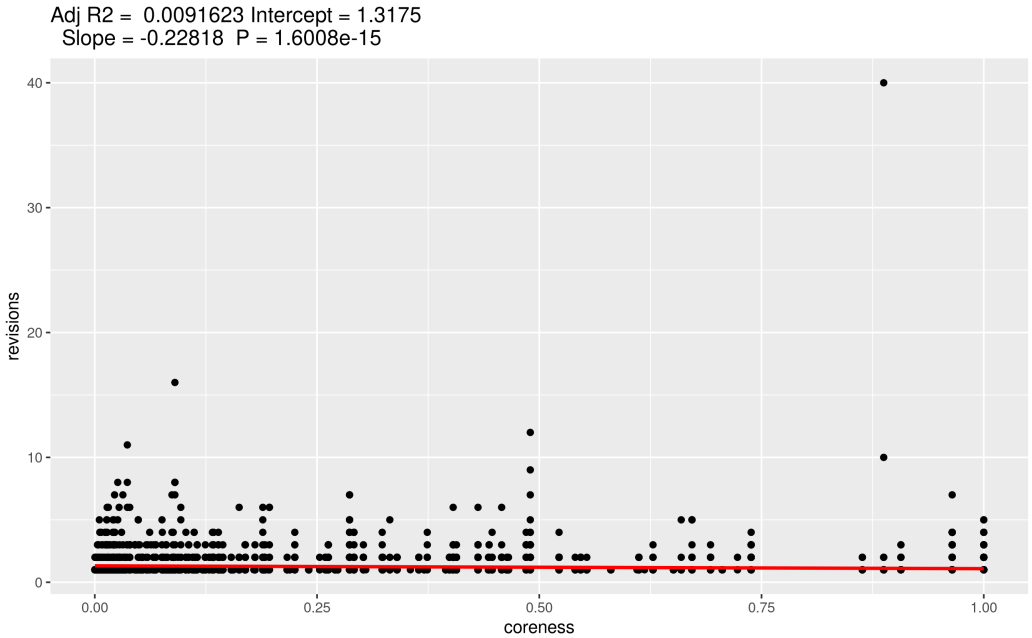


Figure A.10: The scatter plot for the number of revisions under the influence of a developers' coreness-score of FFmpeg analyzed on the mail network with an eigenvector classification.

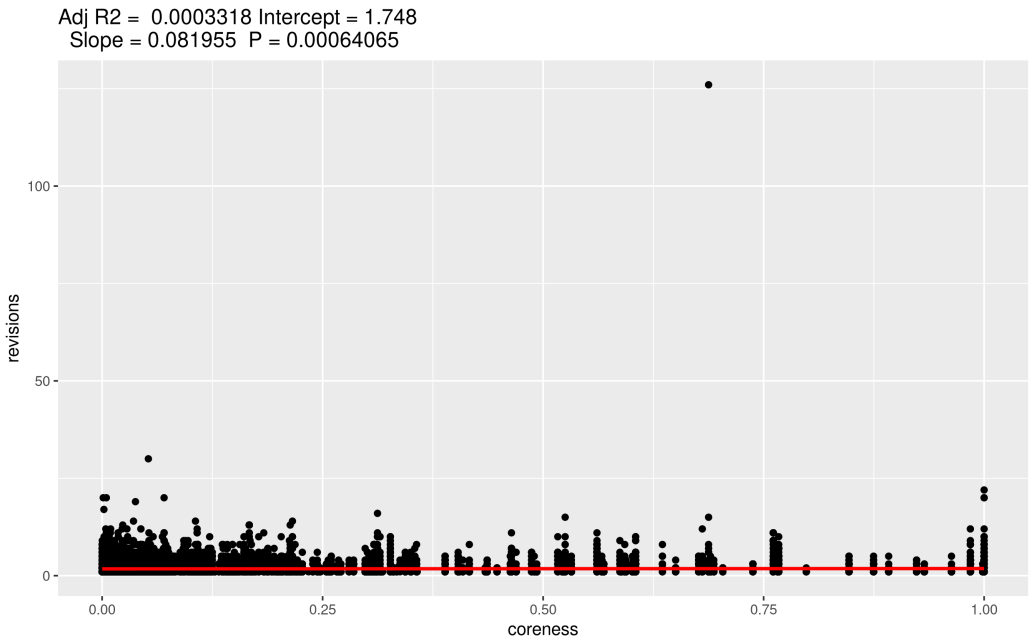


Figure A.11: The scatter plot for the number of revisions under the influence of a developers' coreness-score of U-Boot analyzed on the mail network with an eigenvector classification.

Bibliography

[AJ07] Jai Asundi and Rajiv Jayant. Patch review processes in open source software development communities: A comparative case study. In *Proceedings of the Annual Hawaii International Conference on System Sciences*, HICSS, page 166c. IEEE Computer Society, 2007. (cited on Page 3 and 4)

[BC14] Amiangshu Bosu and Jeffrey C. Carver. Impact of developer reputation on code review outcomes in oss projects: An empirical investigation. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM, pages 33:1–33:10. ACM, 2014. (cited on Page iii, 1, 9, 11, 12, 13, 14, 15, 17, 19, 21, 38, 39, 40, 41, 43, and 45)

[BCB+17] Amiangshu Bosu, Jeffrey C. Carver, Christian Bird, Jonathan Orbeck, and Christopher Chockley. Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft. *IEEE Trans. Softw. Eng.*, 43(1):56–75, 2017. (cited on Page 10)

[BE05] Ulrik Brandes and Thomas Erlebach. *Network Analysis: Methodological Foundations*. Springer, 2005. (cited on Page 5 and 6)

[BLM+06] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D-U. Hwang. Complex networks: Structure and dynamics. *Physics reports*, 424(4):175–308, 2006. (cited on Page 7)

[CGT15] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. Code reviews do not find bugs: How the current code review best practice slows us down. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 27–28. IEEE Press, 2015. (cited on Page 4)

[CWLH06] Kevin Crowston, Kangning Wei, Qing Li, and James Howison. Core and periphery in free/libre and open source software team communications. In *Proceedings of the Annual Hawaii International Conference on System Sciences*, HICSS, pages 118–127. IEEE Computer Society, 2006. (cited on Page 12, 21, and 22)

[DGK12] Mitchel Klein David G. Kleinbaum. *Survival Analysis: A Self-Learning Text, Third Edition*. Springer, 2012. (cited on Page 8 and 9)

[Hec18] Christian Hechtl. The influence of developer roles on contributions to open-source software projects, 2018. (cited on Page 1)

[JAG13] Yujuan Jiang, Bram Adams, and Daniel M. German. Will my patch make it? and how fast?: Case study on the linux kernel. In *Proceedings of the Working Conference on Mining Software Repositories*, MSR, pages 101–110. IEEE, 2013. (cited on Page 9)

[JAHM17] Mitchell Joblin, Sven Apel, Claus Hunsen, and Wolfgang Mauerer. Classifying developers into core and peripheral: An empirical study on count and network metrics. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 164–174. IEEE, 2017. (cited on Page 6 and 9)

[JAM17] Mitchell Joblin, Sven Apel, and Wolfgang Mauerer. Evolutionary trends of developer coordination: A network approach. *Empirical Software Engineering*, 22(4):2050–2094, 2017. (cited on Page 7)

[JMA+15] Mitchell Joblin, Wolfgang Mauerer, Sven Apel, Janet Siegmund, and Dirk Riehle. From developer networks to verified communities: A fine-grained approach. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 563–573. IEEE Computer Society, 2015. (cited on Page 19)

[Job17] Mitchell Joblin. *Structural and Evolutionary Analysis of Developer Networks*. PhD thesis, Universität Passau, Germany, 2017. (cited on Page 5, 6, and 7)

[LFRGBH09] Luis López-Fernández, Gregorio Robles, Jesus Gonzalez-Barahona, and Israel Herraiz. Applying social network analysis techniques to community-driven libre software projects. *Integrated Approaches in Information Technology and Web Engineering: Advancing Organizational Knowledge Sharing*, 1:28–50, 2009. (cited on Page 5)

[LRS17] B. Lin, G. Robles, and A. Serebrenik. Developer turnover in global, industrial open source projects: Insights from applying survival analysis. In *IEEE International Conference on Global Software Engineering (ICGSE)*, ICGSE, pages 66–75. IEEE, 2017. (cited on Page 7 and 10)

[LW05] K.R. Lakhani and R.G. Wolf. Why hackers do what they do: Understanding motivation and effort in free/open source software projects. *Perspectives on Free and Open Source Software*, 1:3–22, 2005. (cited on Page 1)

[Mac67] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297. University of California Press, 1967. (cited on Page 13)

[MBR13] Murtuza Mukadam, Christian Bird, and Peter C. Rigby. Gerrit software code review data from android. In *Proceedings of the Working Conference on Mining Software Repositories*, MSR, pages 45–48. IEEE, 2013. (cited on Page 12)

[MF15] Xavier Blanc Gail C. Murphy Jean-Rémy Falleri Matthieu Foucault, Marc Palyart. Impact of developer turnover on quality in open-source software. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, ESEC/FSE, page 829–841. ACM, 2015. (cited on Page 5)

[OI09] F. Ortega and D. Izquierdo-Cortazar. Survival analysis in open development projects. In *ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, pages 7–12, 2009. (cited on Page 10)

[RB03] Erzsébet Ravasz and Albert-László Barabási. Hierarchical organization in complex networks. *Physical Review E*, 67(2), 2003. (cited on Page 7)

[RGS08] Peter C. Rigby, Daniel M. German, and Margaret-Anne Storey. Open source software peer review practices: A case study of the apache server. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 541–550. ACM, 2008. (cited on Page 9)

[RLM16] Ralf Ramsauer, Daniel Lohmann, and Wolfgang Mauerer. Observing custom software modifications: A quantitative approach of tracking the evolution of patch stacks. In *Proceedings of the International Symposium on Open Collaboration*, OpenSym, pages 4:1–4:4. ACM, 2016. (cited on Page 21)

[RNP+10] Jason Rich, John Neely, Randal Paniello, Christof Völker, Brian Nussenbaum, and Eric Wang. A practical guide to understanding kaplan-meir curves. *Otolaryngology–head and neck surgery : official journal of American Academy of Otolaryngology-Head and Neck Surgery*, 143(3):331–6, 2010. (cited on Page 7)

[SAS10] Ioannis Samoladas, Lefteris Angelis, and Ioannis Stamelos. Survival analysis on the duration of open source projects. *Information & Software Technology*, 52:902–922, September 2010. (cited on Page 10)

[WND08] Peter Weißgerber, Daniel Neu, and Stephan Diehl. Small patches get in! In *Proceedings of the International Working Conference on Mining Software Repositories*, MSR, pages 67–76. ACM, 2008. (cited on Page 9 and 10)

[ZZM16] Jiaxin Zhu, Minghui Zhou, and Audris Mockus. Effectiveness of code contribution: From patch-based to pull-request-based tools. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, pages 871–882. ACM, 2016. (cited on Page 3)

Eidesstattliche Erklärung:

Hiermit versichere ich an Eides statt, dass ich diese Masterarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und dass alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Christian Hechtl

Passau, den 16. März 2020