Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

Jonas Zeuner

BACHELOR THESIS

Exploratory Data Analysis of Code Review Data

Submitted on 2019-08-06

Supervisor:
Michael Dorner
Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander University Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

_____

[CITY], [DATE]

License

_____

[CITY], [DATE]

Abstract

The code review process is well established and an essential part during the development of open source software. To establish code reviews in open source projects, the process of code review is often supported and monitored by additional review tools like Gerrit. This thesis analyzed review data from Gerrit for some open source projects (i.e. Gromacs, Go and Typo3) with the main focus on the observation of the impact of highly active developers during the review process. Therefore, indicators to identify these active developers during the review process were introduced and used to prepare a selection of active developers. Based on this selection their impact during the review process was analyzed. By applying an appropriate threshold on the introduced indicators, the size of the selected highly active developers (top-k developers) can be customized. The final findings showed, that there is a small selection of top-k developers that have a strong impact on the review process of the open source project they participate in (e.g. 1% of the 2492 contributors in the Typo3 project provide about 70% of the measured review content). In addition to this analysis, the preparation of the review data from Gerrit and a short descriptive overview are presented.

Acknowledgements

I would like to thank my supervisor Michael Dorner for introducing me to the research topic, for his expert advice, his encouragement and his support through this project. He supported me greatly and was always willing to help me.

I want to acknowledge the Open-Source-Software chair from Prof. Dr. Dirk Riehle and Prof. Dr. Dirk Riehle himself for his always motivating lectures to all connected Open-Source Topics. For their well-organized thesis process and the friendly atmosphere.

Lastly, I want to thank my family, for always supporting me. You are always there for me.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

JSON	JavaScript Object Notation
NaN	Not a Number
EDA	Exploratory Data Analysis
RQ	Research Question

1 Introduction

1.1 Original Thesis Goals

It was the original goal of this thesis to generate a descriptive overview of the provided data set of code review data from Gerrit by using an exploratory analysis. To analyze the data set, the data should be prepared. This preparation should be a foundation for research and the reuse of the data set in the future. In addition to the analysis, the data base should be used to model networks for different open source projects with python [1] and networkx [2]. Potential resulting hypotheses and theories, from findings during the exploratory analysis and the creation of networks, should be discussed.

1.2 Extensions to Thesis Goals (if any)

After the process of the exploratory analysis of the data set, the thesis goal was extended. Due to the general overview, several theories and hypotheses have emerged. This thesis decided to focus on the impact of the top-k developers during the review process of different open source projects.

By the exploratory analysis of the data set and the created networks, there were developers observed that occurred more frequently than others. These developers were very active participating during the review process. This led to the hypothesis that there is a small selection of developers that have a strong impact and important role in the review process in the project they participate in.

The thesis goal has been extended to observe the impact of these top-k developers in a selection of the provided data set.

Therefore, the top-k developers needed to be identified. To identify them, multiple indicators were introduced. On the one hand indicators that used statistical measures of the review data base and on the other hand indicators that used network theory to identify the top-k developers.

Finally, the resulted findings were described and discussed.

2 Research Chapter

2.1 Introduction

To guarantee quality during the development of software, it is common practice to perform code reviews during the development. The code review process is well established and an essential part during the development of open source software. To establish code reviews in open source projects, the process of code review is often supported and monitored by additional tools. One of these tools is Gerrit [3]. This tool is used by many open source projects (for example Chromium, Android, etc.) and allows a tool-based review process. Within this thesis a collection of code review data from Gerrit was analyzed. The original thesis goal focused on the preparation and an exploratory analysis of the provided data.

During this exploratory data analysis, a selection of developer's names has occurred more frequently than others. These developers had a great impact on different measured sizes of the code review process. This conspicuity emerged in the hypothesis that a small selection of developers dominates the code review process. With the motivation to better understand the analyzed code review process of the different open source projects this thesis decided to extend the original thesis goal.

To observe the situation and to offer a deeper insight into the review process of three open source projects (i.e. Gromacs, Go and Typo3), this thesis focused on the observation of the behavior of active developers and their impact on the different review processes. To generate this understanding a field study approach was chosen to analyze the code review data from Gerrit.

In detail, this thesis's contributions are the following:

- Preparation of code review data from Gerrit
- Exploratory analysis of the prepared data
- Field study on the impact of the most active developers (top-k developers)

To guide the field study a general research question was introduced. This question was divided into sub-questions that build on each other and establish the analysis of the general question.

The performed field study in this thesis is structured based on these research questions.

2.2 Related Work

Because this thesis first used exploratory data analysis to explore collected data from Gerrit and then extended the thesis goal to analyze the impact of top-k developers during the review process in different open source projects, this section will be spited into two parts.

Analysis of Code Review Data from Open Source Projects

In 2013 Mukadam et al. [4] generated an overview of Gerrit data for the open source project Android. They described the extraction, preparation and available information from the peer review data of the Android project from Gerrit.

The following works mainly focused on further research that also used data collected from the Gerrit tool for their analysis. They provided a deeper insight into the different data sets and the data provided by Gerrit.

Mishra and Sureka [5] collected review data from Gerrit and another review tool Rietveld. With these data sets they analyzed a collection of hypotheses about the effort and contribution of reviewers for different patch sets.

In 2016 Kitagawa et al. [6] used data provided by Gerrit to analyze the behavior of the participating reviewers, by use of theoretical game modeling.

Lipcak and Rossi [7] used Gerrit data as additional source to analyze different approaches for productive reviewer recommendations.

The data, provided by the Gerrit API, was also used in 2018 when Spadini et al. [8] introduced their own Gerrit crawler ("GerritMiner"). With this crawler they generated a data set that they used as a part of their analysis of automated testing during the code review process.

Beside the analysis of Gerrit data, there are many works that focused on analyzing the review process itself.

Rigby et al. [9] generated an overview of the peer review process for open source projects, when the review process was mainly arranged by email exchange.

Compared with that, modern works focus more on increasing the quality of the review process, like Kononenko et al. [10] that provided a better understanding of the review process and possible risks from that process.

Other modern works focus on tool-based code reviews. Works like Bacchelli and Bird [11] showed how to improve the review process and the benefits of using these tools.

Identification of Top-k Developers in Open Source Projects

Because there is no related work that focuses on the impact of active developers during the review process, this part shows work that focuses on identifying important users in a network structure. Code review data can be represented as a network structure, the identification of potential top-k developers can benefit from this representation and can use network theory to support the identification process. Again, there is no other work that used Gerrit data to identify important users. As many scenarios can be represented in a network structure this field is analyzed by a lot of other research works.

Many of these modern works mainly focus on inventing new methods that are based on different fundamental network theory approaches to identify important or influential nodes in a network.

One of these methods is the so called "Gaussian kernel method" from Sharma et al. [12] that is based on the degree, closeness and betweenness of nodes in social media networks to identify most influential nodes. This work is based on different research works like Kempe et al. [13] that organized and performed an experiment on collaboration networks.

The work about the "Gaussian Kernel Method" also reported about other works, like Scripps et al. [14] and Shetty et al. [15] that used different approaches on the network structure, that both performed well. The first of these works was based on network communities, while the other approach focused on information theory with statistical techniques.

Another method is the so called "CS-TopCent" method from Mahyar [16] that used multiple different centrality metrics to identify the most central nodes.

In additional to the above-mentioned methods, there are many other approaches to identify important or influential nodes in networks. Like Nema et al. [17] that used the clustering effect and classifications of the different networks for identification.

But most of the modern works used a similar set of centrality measures, other similar works are Wen et al. [18]. This work focused on the information dissemination in online social networks and therefore identified popular users in a network structure by their degree, core, betweenness and community bridges they are engaged in.

Zhao et al. [19] used the connection of the nodes to their neighbors to introduce a method that is based on local centrality.

Huang et al. [20] also used centrality indicators but focused on ego network structures to not only identify key nodes but also to identify the role of the node in the network. This work additionally pointed out, that it is better to use indicators with a strong correlation to identify important individuals. These indicators again showed a good performance of the indicators in heterogeneous networks.

Another approach, also used in this thesis, is shown by Ghoshal and Barabási [21] in their work. They showed that the pagerank algorithm from Google is a good indicator to simply identify a small selection of so called "super-stable nodes" that are nothing else than the key users in a network structure. They also recommend to not over complicate the identification of these nodes because the algorithm already serves very good results.

2.3 Research Question

This thesis analyzes the review process of open source projects with a special focus on the impact of individual developers within the open source project community.

The research question is formulized as:

"What is the impact of highly active developers (top-k developers) during the review process?"

The analysis is structured along the following topics, to introduce indicators that can be used to identify those developers which actively participate and have a great impact on the review process (i.e. top-k developers). Next, identifying the top-k developers and representing their impact during the review process on the different open source projects.

Therefore, the overall research question is split up into three sub-questions.

RQ 1: "What is the impact of the top-k developers during the review process?"

RQ 1.1:	How to identify active developers, what are useful indicators?
RQ 1.2:	Who are these active developers?
RQ 1.3:	What is their impact on the review process they participate in?

2.4 Research Approach

The first step in this thesis is based on an "Exploratory Data Analysis" (EDA) approach, first introduced by John Tukey in the 1970s, to analyze the provided code review data set from Gerrit. EDA is used to generate a general overview of the multi-dimensional data and to provide an orientation for further analysis by visualization of the provided data set (see [22]).

As result of this exploratory data analysis a hypothesis has emerged, that there is a small selection of very active developers with a significant impact on the review process of open source projects.

To observe this hypothesis by analyzing the research questions described in the Research Question section 2.3 above, a field study approach was chosen.

The use of a field study approach was based on the decision table in Stol and Fitzgerald [23] and the fact that the analysis is oriented on a specific real-world setting with the goal to analyze a specific software engineering phenomenon (i.e. the impact of top-k developers during the review process of open source projects). This is also supported by the data used for this thesis (see 2.5 Used Data Sources), which is collected in a natural setting (i.e. Gerrrit) and based on the fact that there was no aim to change the research setting or manipulate any variables.

Therefore, this method allows to observe a particular system (i.e. open source projects) by referring to a specific setting (i.e. review process) which "offers maximum potential for capturing a realistic context" (see [23], p.11). This fits with the main motivation of this thesis to obtain a rich description of the behavior of the community developers participating in the review process of different open source projects.

To observe the review process and to study the impact of the top-k developers during the review process (RQ 1) of different communities (i.e. open source projects: Gromacs, Go and Typo3) these top-k developers had to be identified.

Therefore, different indicators were introduced and used to identify active developers. On the one hand, statistical and descriptive sizes measured in the data set are used to identify the developers. On the other hand, a network analysis is used to support the selection. Throughout a combination of both methods the selection of the top-k developers has been achieved.

Based on this selection the impact of these top-k developers has been estimated by their contribution to the projects in comparison to the overall project community.

The resulting description of the communities (i.e. Gromacs, Go and Typo3) are based on the description and discussion of these estimations.

2.5 Used Data Sources

2.5.1 Data Source Origin

The main data source, used within this thesis, is data crawled by Gerry, a crawler for code review data from Gerrit instances [24].

The collection analyzed within this thesis consists of twelve open source projects, differing in the application domains of the projects and the number of crawled reviews. The available data covers different time periods for each project.

Within this thesis, out of the available data set of twelve projects, a selection of three projects was chosen, to analyze the extended thesis goal: Gromacs, Go and Typo3. Table 1 lists these open source instances and their different attributes (also see [25]).

This selection was chosen in order to establish the analysis goal with a smaller data set (i.e. Gromacs), which differs from the other open source projects in the domain it is located in and the number of reviews and reviewers. The two data sets (Go, Typo3) were chosen to verify the results achieved with the first data set. Go and Typo3 represent different domains and larger open source projects. For further representations the Go project was chosen, because of the limited time period of this project and the representative size of the collected data.

Open Source Project Name	Available Review Data	Number of Reviews	Number of Reviewers	Domain
Gromacs	2715 days 2011 - 2017	7547	156	Simulation Framework
Typo3	2861 days 2011 - 2017	30530	1371	Content Management System
Go	1508 days 2014 - 2017	33851	2492	Programming Language

Table 1: Overview of the used open source instances and their review data

The crawler Gerry provides review data extracted from two different sources. The first source is extracted via the search-based change access of Gerrit. Unfortunately, this method provides data, which is not always complete. Therefore, this method has been excluded due to its volatile behavior. The second approach, via direct access of Gerrit, provides stable and complete data sets and is used within this analysis (see [25]).

The crawler provides each review as a single JSON file, which contains all data of this individual review. The largest used data set, the Go project, comprises about 30.000 review files.

In order to facilitate the analysis process, during which each file has to be opened and read multiple times, a dedicated data preparation of the available data was required. This data preparation (see 3 Elaboration Chapter) included data cleaning, handling of missing data and a reorganization of the data for analysis purposes.

2.5.2 Preparation for Network Analysis

To use the re-organized and cleaned data for further network analysis, the provided data was used to create multiple graphs in python. This chapter describes the creation of these graphs. The fundamental basics of network theory used for this creation are described in the section 3.1.2 Network Theory in the elaboration chapter.

The graphs created for further analysis were based on the exchange of messages between users participating in a code review. The nodes in these graphs represent users that send or receive messages during the code review. The edges describe the communication via messages between two users. These edges were directed based on the information, who created the message and who received the message. There were no multiple edges used, if a user sends multiple messages to another user. Instead, the edges were weighted by the number of exchanged messages between those users. Therefore, the networkx (see [2]) graph type DiGraph was used.

Using this approach on the full data set of the Gromacs project, results in a large graph with about one hundred-fifty nodes and about three-thousand edges. As the review data of the

Gromacs projects comprises data from 2011 to 2017, the graph is a convolution of time depended data and therefore not useful for further analysis on the research question. In the other two projects (Go, Typo3), where there are even more users participating, this effect is even bigger.

To avoid this problem and to minimize the impact of this effect and to clearly extract, if a user was only active for a limited time period, the large graph was split-up on a yearly basis.

```
# Input: reviews, messages

reviews = {review: reviewers}
messages = [author, date, review]

graphs = {year: G(V,E)}

for author, date, review in messages:
    if author not in graphs[date.year].V:
        # Create Node for Author
        graphs[date.year].V.add(author)

    for reviewer in reviews[review]:
        if reviewer not in graphs[date.year].V:
            # Create Node for Reviewer
            graphs[date.year].V.add(reviewer)

        if (author, reviewer) not in graphs[date.year].E:
            # Create Edge for Author -> Reviewer
            graphs[date.year].E.add((author, reviewer), weight=1)
        else:
            # Increase number of messages for Edge Author -> Reviewer
            graphs[date.year].E[(author, reviewer)].weight += 1

# Output: graphs
```

Figure 1: Algorithm to create the communication graphs

The basic structure of the final algorithm, to create the different graphs for each year, is described in pseudo code Figure 1.

To create the graphs the networkx library for python was used. The library provides an own data structure for graphs, that allows a simple way to use graphs in python and enables analysis supported by this graph structure.

The resulting graphs were exported in the graphml structure (see [26]). The graphml format does not use an own syntax to store the graph structure like other graph file formats, instead it is based on the XML format. This format allows to load and to store the graph structure for further analysis.

To create a visualization of the different communication graphs, the python-igraph library (see [27]) was used. With the help of the Fruchterman Reingold algorithm (see [28]) the layout of the graphs was generated. This algorithm creates force-directed graph drawings, by considering a force between two nodes. The force is represented by the weight of the edges between those two nodes.

For the presented graphs in the Figure 2, Figure 3 and Figure 4 , only the main structure is shown, isolated nodes and sub graphs with up to three nodes were ignored.

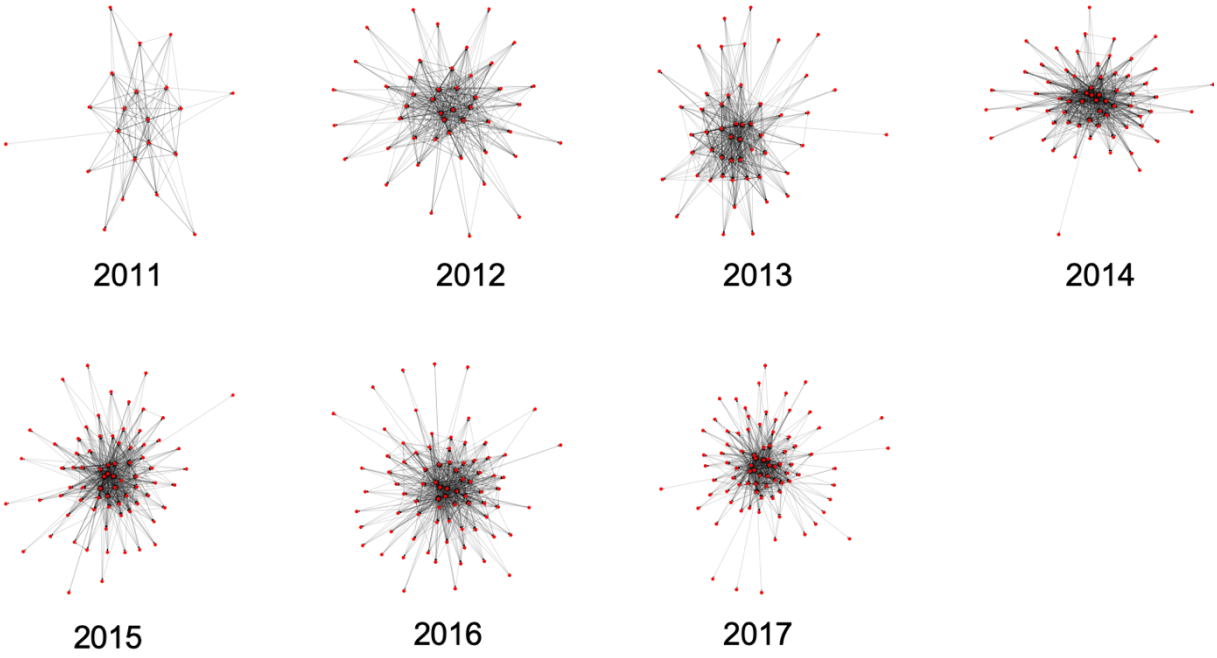

Figure 2: Fruchterman Reingold layouted graphs for the Gromacs project (from 2011 to 2017)

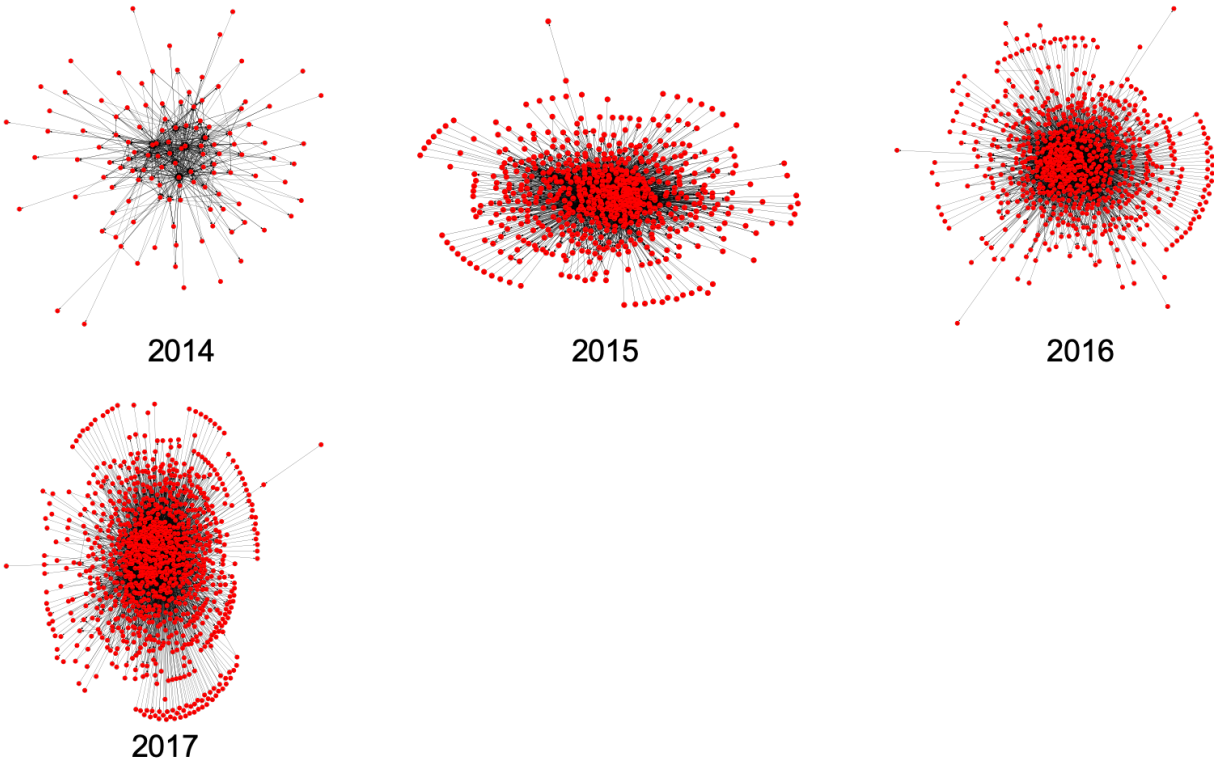

Figure 3: Fruchterman Reingold layouted graphs for the Go project (2014-2017)

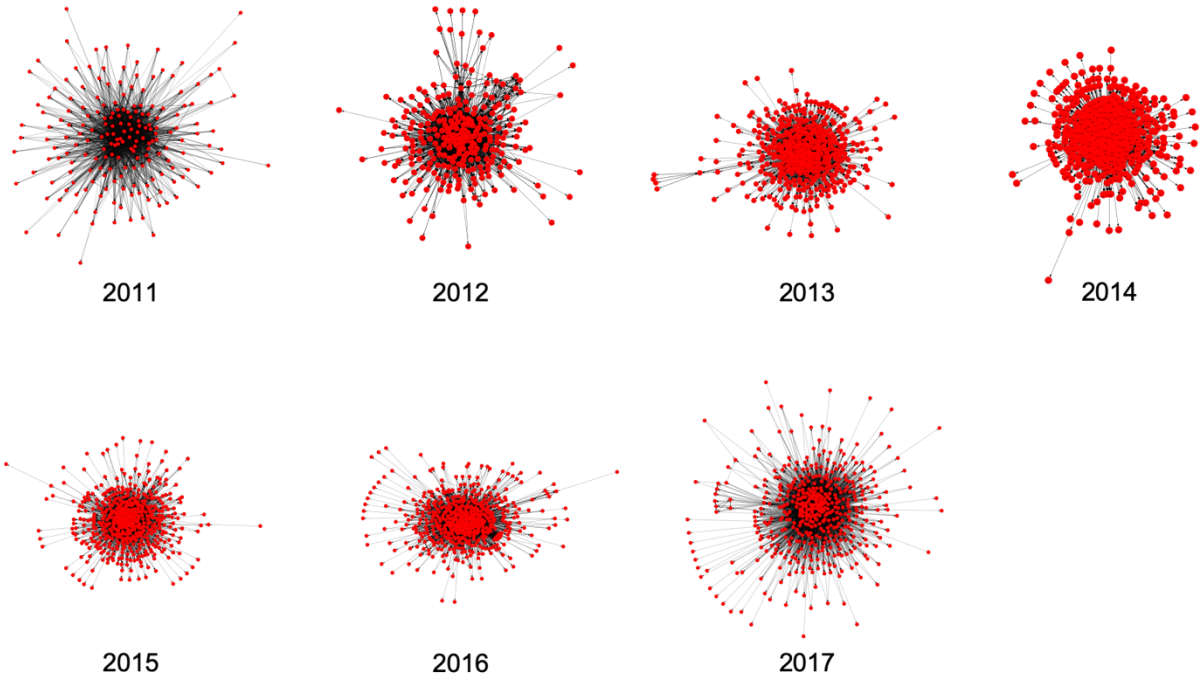

Figure 4: Fruchterman Reingold layouted graphs for the Typo3 project (2011-2017)

Already from these visualizations of the different graphs it is clear to see that the Gromacs project is by far less complex and is smaller than the other two projects.

2.6 Research Results

2.6.1 Indicators for Identification of active Developers

To answer the research question, the most active developers have to be identified. To achieve this purpose, multiple indicators will be introduced. These indicators support answering the research question 1.1 and have the potential to identify the important developers within a project.

Within the scope of this thesis two different approaches were considered (statistical measures, network theory).

Statistical Measures

The first approach is based on multiple statistical measures, these measures can be used as indicators to identify the active developers in an open source project. These measures use the code review data and were therefore chosen by this thesis to represent the time and the effort spent on the project by each single developer.

To create this representation the following measures were chosen:

- F_{CR}: Fraction of created reviews
- F_{CM}: Fraction of created messages
- F_{RR}: Fraction of reviewed reviews

The different fractions use the number of the respective measure (reviews, messages, reviewed reviews) normalized by the total number.

These indicators were chosen because they represent the participation during the review process. As explained in the sections about code review and tool-based reviews, the review process is a major component of the software development process and therefore results obtained for this process are a good representation of the overall contribution in an open source project.

Network Theory

The second approach is based on network theory. The theoretical foundation used for this approach is summarized in the 3.1.2 Network Theory.

In this case, the indicators refer to the structure of the different communication networks for the individual projects, that were created in the section 2.5.2 Preparation for Network Analysis. To identify the most important developers, this approach uses the position and the behavior of communication during the review process as indicator. In contrast to the first approach, where the main idea was to use the quantity of participation as indicator.

Within this second approach, different measures from network theory are derived as indicators. The first part of these measures is based on classical centrality indices from network theory (see [29]). Because these indices create a ranking of nodes, they are able to identify most central nodes. Therefore, these rankings can be seen as indicators to identify important nodes in a network structure. For the goal of this thesis the following centrality indices will be used:

Degree centrality

The degree of a node is the number of direct neighbors adjacent to this node. Degree centrality (see [30]) builds on this fact, it is a basic measure that describes how many direct contacts the node has and therefore describes the centrality of this node. To calculate this centrality, the degree of the node is normalized by $n - 1$:

$$C_D(v) = \frac{\deg(v)}{n-1}$$

This measure describes the direct influence of a node and its access to information from direct neighbors. By this it is a useful indicator for the active developers and describes if a node has a strong connection within the network.

Betweenness centrality

The betweenness centrality measures (see [31]) the proportion of the shortest paths in the network to the shortest paths passing through the node v. This measure is introduced by $C_B(v)$ where:

$$C_B(v) = \sum_{s,t \in V} \left[\frac{\sigma(s,t|v)}{\sigma(s,t)}\right] / ((n-1)(n-2))$$

"Where V is the set of nodes, $\sigma(s,t)$ the number of shortest (s,t) paths, and $\sigma(s,t|v)$ the number of those paths passing through some node v other than s,t" [32].

The algorithm of the networkx library normalizes the values by $1/((n-1)(n-2))$ for directed graphs. The measures $C_B(v)$ describe the control of information and are therefore another reasonable indicator for the active developers.

Eigenvector centrality

The eigenvector centrality (see [29]) uses the adjacency matrix A of the graph G to calculate the eigenvalue λ for the v-th node element in the vector x in the equation:

$$Ax = \lambda x$$

The eigenvector centrality C_E is where:

$$C_E(v) = \{ \text{return } \max(\lambda) \text{ for node } v \}$$

For any given node v, the v-th eigenvalues are calculated, with the additional requirement that all the entries in the eigenvector have to be non-negative. The "Perron Frobenius theorem" implies that the greatest eigenvalue is the searched eigenvector centrality (see [33]).

The algorithm used by the networkx library uses a slightly different approach to calculate the eigenvector centrality, but the main aspects are still the same.

The eigenvector centrality measure describes the centrality of a node v by the centrality of its neighbors and therefore its influence in the network. It can therefore be used as another indicator for active developers.

It is possible to use additional centrality indicators for the identification of the active developers in a network structure. For the analysis within this thesis the centrality indicators have been restricted to the three indicators described above. However, for validation of the obtained results a comparison to another indicator (i.e. Google's pagerank algorithm) has been performed.

PageRank Algorithm

Pagerank (see [34]) is an algorithm invented by Google Search. It was originally used to rank web pages by the relative importance, to deliver good results for their search engine. The algorithm operates on a network structure and computes a ranking for each web page, depending on the structure of the incoming links. In general, the web pages are represented by nodes and the incoming and outgoing links are their edges. The algorithm can therefore be used as indicator for important nodes in a network structure. The pagerank ranking can be introduced by $R(u)$ where:

$$R(u) = c \sum_{v \in B_u} \frac{R(v)}{N_v} + cE(u)$$

u is the node, for which the ranking is calculated. B_u describes the set of nodes that u has outgoing edges to. $N_u = |F_u|$ is the number of edges from u and c is a factor used for normalization. $E(u)$ is a vector over the nodes that matches to a personalized nonzero ranking, if not given, networkx replaces this vector by an uniform personalized vector.

The implementation in networkx uses a recursive approach which does not always converge and therefore stops after a maximum amount of iterations. From the abstract description of the ranking, it is obvious that the implementation operates on a directed graph structure.

2.6.2 Identified Top-k Users in Open Source Projects

To identify the most active developers, the different indicators, as introduced in the section above, have to be calculated. With the resulting rankings a selection of active developers can be identified. With this selection research question 1.2 is answered.

To identify these developers with the different indicators, the whole available data set was divided by the years of collected data. This ensures that both described approaches, the statistical measure and the network theory sizes operate on the same set of data. The networks were already partitioned for the different years during the creation as described in section 2.5.2 . This annual filter and the fact that all indicators are normalized, allows the comparison of the different results.

As next step, the different indicators are calculated, the general procedure used in this thesis is represented in the pseudo code Figure 5.

```
# Input: projekt

project # Project Name

(reviews, messages) = getTables(project) # {year: (ReviewTable, MessageTable)}
graphs = getGraphs(project) # {year: Graph}

TopKs = []
GeneralOverview = []

for year in graphs.keys():
    GeneralOverview += getGeneralOverview(graphs[year])

    centrality_scores = calculate_Centrality_scores(graphs[year])
    r_scores = calculate_R_scores(graphs[year])
    fraction_scores = calculate_Fraction_scores(reviews[year], messages[year]

    TopKs += getTopKUser(centrality_scores, r_scores, fraction_scores, THRESHOLD)

TopKDeveloper = getFinalTopKUser(TopKs)

# Output: TopKDeveloper, GeneralOverview, TopKs
```

Figure 5: Algorithm for Identification of Top-K Developers

First of all, this calculation results in a general overview of basic direct indicators that describe the participation in the projects for the different years (these indicators were introduced in the section 3.1.2 Network Theory).

The following overview Table 2 shows the number of nodes n, the number of edges m and the density d of the different communication networks for the Go, Gromacs and Typo3 projects.

Year	**Go Project**			**Gromacs Project**			**Typo3 Project**		
	n	m	d	n	m	d	n	m	d
2011	-	-	-	21	184	0.438095	191	3599	0.099173
2012	-	-	-	47	639	0.295560	265	5155	0.073685
2013	-	-	-	51	719	0.281961	449	8548	0.042495
2014	123	949	0.063241	65	984	0.236538	526	9559	0.034615
2015	601	6529	0.018106	75	910	0.163964	508	9741	0.037821
2016	745	7570	0.013657	80	952	0.150633	512	10493	0.040106
2017	1027	8996	0.008538	84	876	0.125645	458	10472	0.050032

Table 2: Direct indicators for each open source project split in years

Beside the above shown general measures, for the different networks, the calculation derived all scores for each developer, representing his individual participation within the project on a yearly base. For ease of clarity, in the following, the obtained results for the Go project will be shown (overviews and figures for the other two projects – Gromacs and Typo3 – are included in the appendix). All three projects show a comparable behavior.

The full overview of the cumulative sum of the different scores (fraction of created reviews, fraction of created messages, fraction of reviewed reviews, degree centrality, betweenness centrality, eigenvector centrality, pagerank) calculated for the Go project for the years from 2014 to 2017 for each user is represented in the Figure 6 (also see Appendix A).

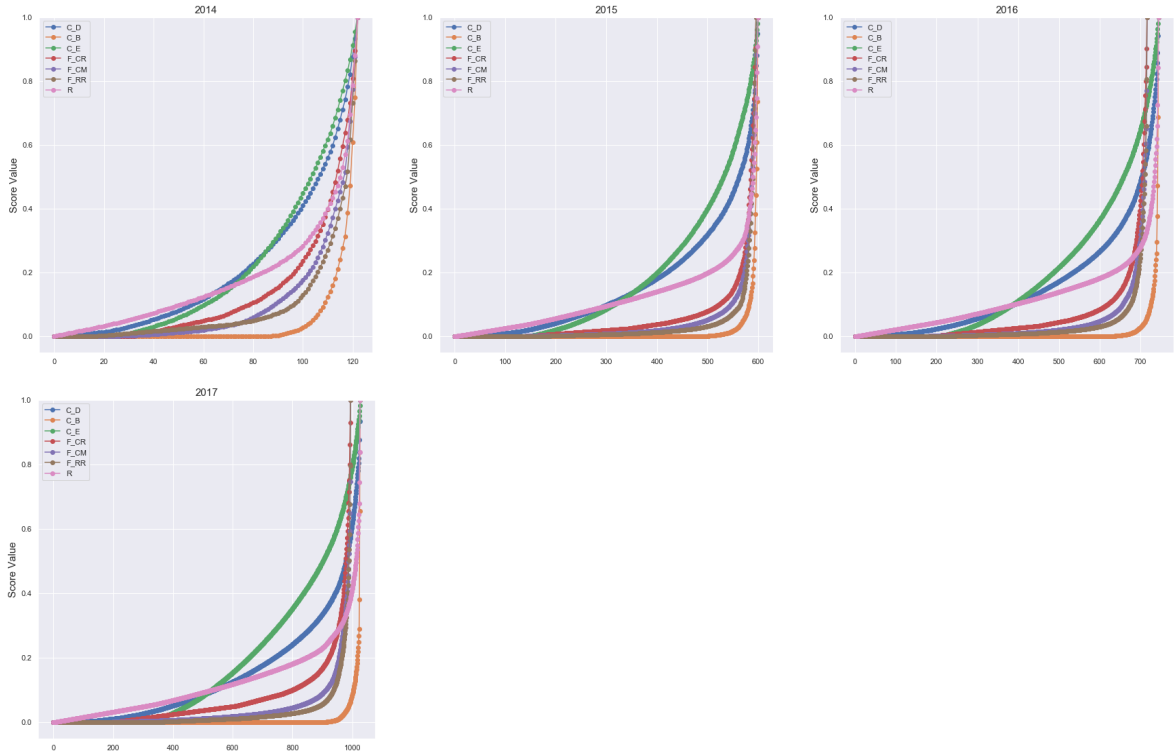

Figure 6: Cumulative sums of all seven indicators versus the sorted (by score contribution) active developers for the Go project from 2014 to 2017

As seen from this figure, all approaches (fraction of created reviews, fraction of created messages, fraction of reviewed reviews, degree centrality, betweenness centrality, eigenvector centrality, pagerank) show a similar behavior, the contribution is not equally distributed, a small fraction of developers provide the main contribution to the scores.

To better display this effect, the Figure 7 focused only on the cumulative sum of the degree centrality for the Go project in the year 2015.

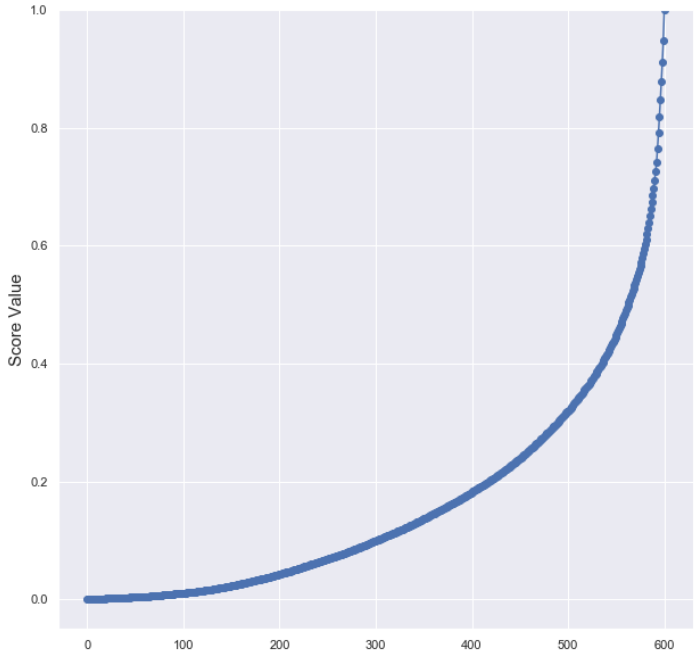

Figure 7: Cumulative sum of degree centrality versus the sorted (by score contribution) active developers for the Go project in 2015

By representing the network theory approaches (i.e. degree centrality, betweenness centrality, eigenvector centrality, pagerank) over the years of 2014 to 2017 in a pair plot (see Figure 8), a correlation between the different approaches can be observed.

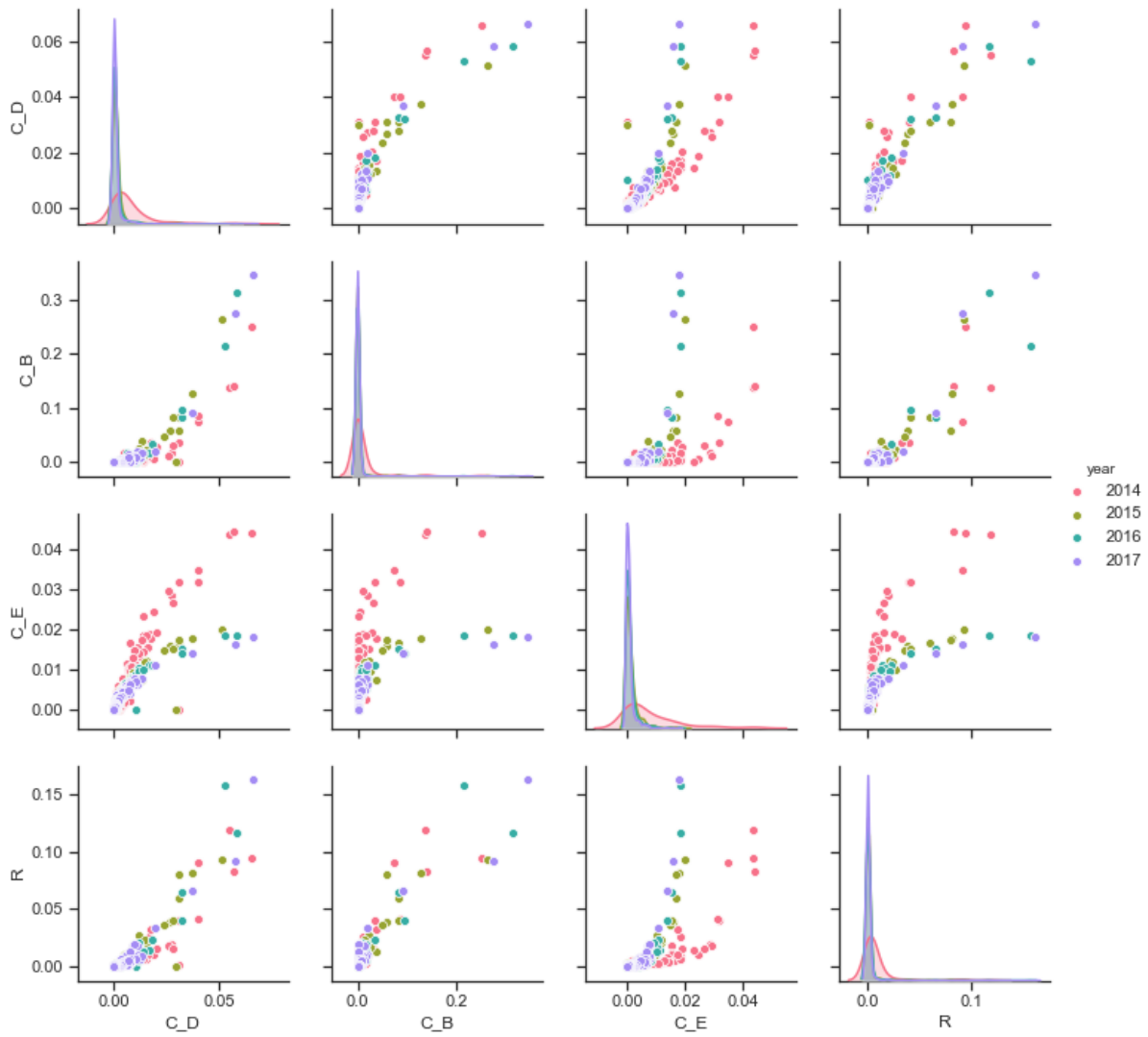

Figure 8: Pair Plot (multiple scatter plots) of the network theory indicators in the Go project from 2014 to 2017

The observed behavior is similar for all projects, there is a correlation between the different approaches to calculate the scores. For example, the correlation values for the different approaches in the Go project for the year 2015 is represented in Table 3 (comparable correlation values can be calculated for the other projects and given periods).

	C_D	C_B	C_E	R	F_{CR}	F_{CM}	F_{RR}
C_D	1	0.846	0.867	0.916	0.779	0.866	0.909
C_B	0.846	1	0.646	0.893	0.697	0.640	0.866
C_E	0.867	0.646	1	0.790	0.791	0.646	0.803
R	0.916	0.893	0.790	1	0.790	0.807	0.986
F_{CR}	0.779	0.697	0.791	0.790	1	0.632	0.862
F_{CM}	0.866	0.640	0.646	0.807	0.632	1	0.796
F_{RR}	0.909	0.866	0.803	0.986	0.862	0.796	1

Table 3: Correlation coefficients of the different identification approaches in the Go Project from 2015

Applying the different approaches, for each approach the scores per developer can be calculated. To select the final top-k developers, a threshold has to be applied. For a first selection of the top-k developers, a threshold of 80% of the respective score has been used in this thesis. In the next section 2.6.3 , there will be a closer look at choosing an appropriate threshold and its effect on the number of top-k developers and their impact on the project.

Nota bene, applying higher threshold only generates sub sets of the identified top-k developers identified in this section.

The following heatmaps (see Figure 9) show the scores for the three approaches (i.e. degree centrality: C_D, pagerank: R, fraction of reviewed reviews: F_{RR}) of the selected top-k developers of the Go project, separated by the years 2014 – 17 (also see Appendix A)

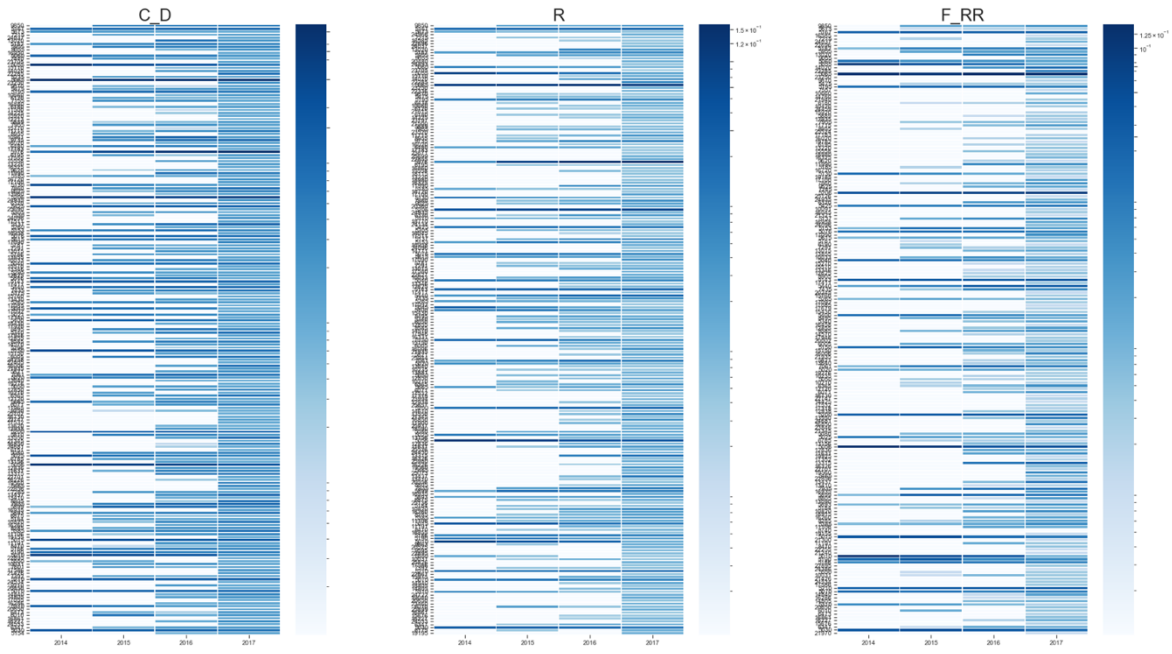

Figure 9: Top-k developers and their different indicator scores for the Go project

It can be observed that some developers were active over the whole period, others contributed only in parts of the period. The color coding represents the score value.

Comparing the top-k developer sets derived by the different approaches, the intersection set is much greater than the difference set, as expected from the high correlation coefficients of the different approaches.

Therefore, this thesis uses as selection for the final set of top-k developers for a project the intersection of the different approaches over the given period 2014 – 17 (Go project).

Applying this approach to the three projects results in a list of developers for each project, which represent the top-k developers. The resulting size and the percentage of top-k developers to the total number of participators in the different projects are represented in Table 4.

	Number of Top-k Developers	**Total Number of Developers**	**Percentage of Top-k developers**
Gromacs	12	156	7.69 %
Go	100	1371	7.29 %
Typo3	79	2492	3.17 %

Table 4: Overview of the number of top-k developers (80% threshold) in comparison to the total number of developers for the three projects (Go, Gromacs, Typo3)

For the Go project the list of these top-k developers is presented in the Table 5 (the top-k developer lists for the other projects can be found in Appendix B).

9850	5167	5182	5955	13020	5060	5055	14020	5065
5475	5195	12530	7155	6545	6576	7955	13460	11715
10961	9735	6480	5976*	11990	5130	9525	7860	5206
5170	6320	14596	5425	5137	5153	5076	5615	13015
10033	5040	5846	5143	12640	5440	7435	7050	5300
5446	5265	5021	5045	5400	8495	5340	8585	5480
5150	6005	5930	8065	7061	5020	5335	10107	12850
6365	5665	6071	8481	5515	5255	5200	5810	11900
5080	5025	5056	11631	13315	13437	7455	7935	5899
5893	5036	10820	8285	5385	11106	6173	5015	11191
5010	5186	5190	10031	5210	5070	5310	10715	7250
5030								

*Table 5: Identification number of the top-k developers in the Go project (Project Bot marked with *)*

2.6.3 Impact of the Top-k Developers on Open Source Projects

With the selected set of top-k developers for the different open source projects from the previous section, their respective impact can be estimated.

One possible option to estimate the impact and the participation of these developers is by calculating different measures in the data set (i.e. the number of created reviews or comments).

The following Figure 10 displays four of these indicators. For each indicator the contribution of the top-k developers is compared to the overall contribution of all participators.

The top-k developer contribution is marked blue, the contribution of all developers is marked orange.

Again, a similar behavior can be observed for the Gromacs and Typo3 projects (see Appendix C).

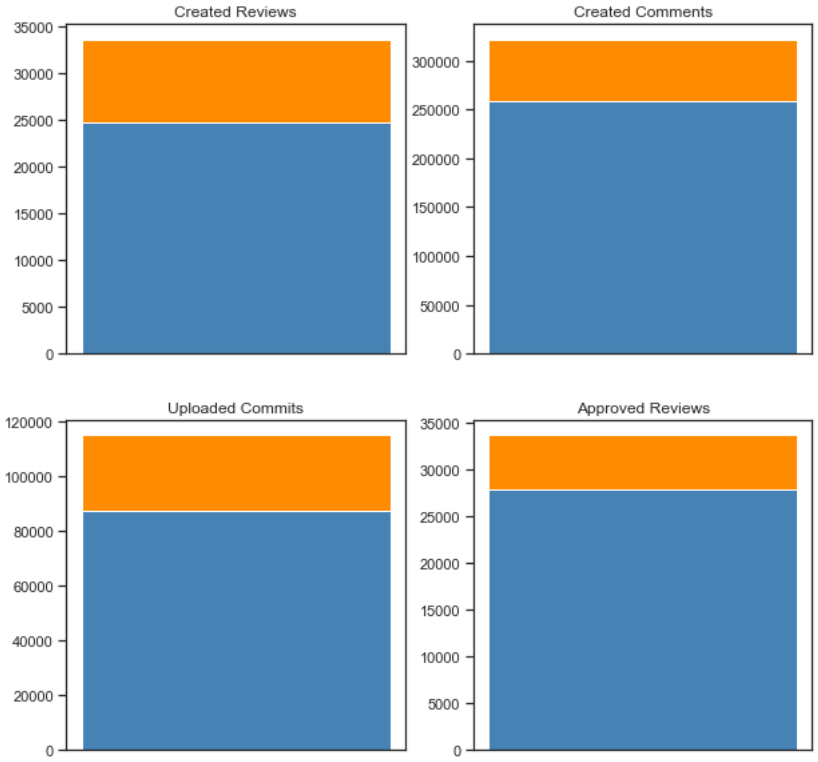

Figure 10: Impact of the top-k developers in the go project (top-k developer contribution is marked blue; contribution of all developers is marked orange)

The following pie-charts (see Figure 11) show distributions of the participation between top-k developers and the whole project developers (top-k developer: blue, all developers\top-k developers: orange). The figures for the other projects can be found in the Appendix C.

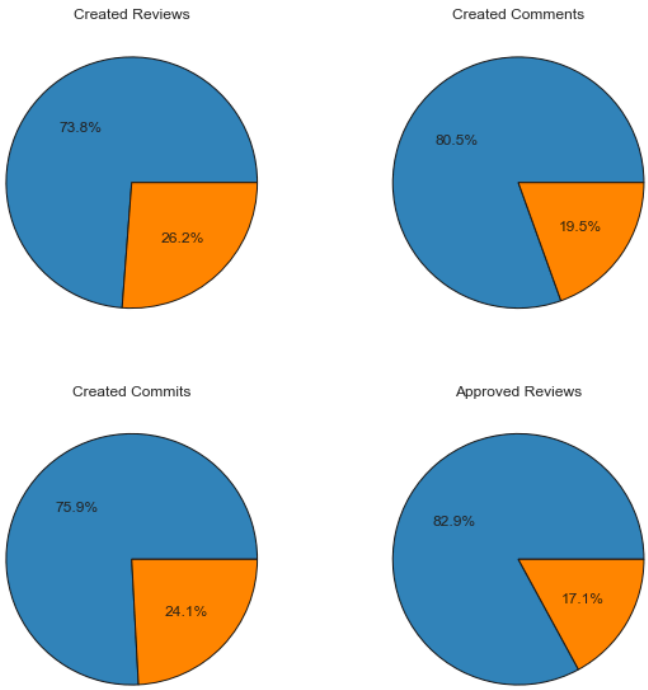

Figure 11: Percentage of impact (measured in created reviews, created comments, created commits and approved reviews) of top-k developers vs. remaining developers in the Go project

During the exploratory data analysis, it became obvious that the number of created comments correlates with the number of created reviews. Therefore, it was predictable, that a developer who created many reviews was also active in creating comments.

The following scatter plot Figure 12 shows this effect, by plotting the fraction of created reviews against the fraction of created comments per developer. The figure for the Go project shows a linear correlation with a correlation coefficient of 0.605 (including all data points) and 0.936 (excluding outliers, namely the bot), similar to the values in the other projects.

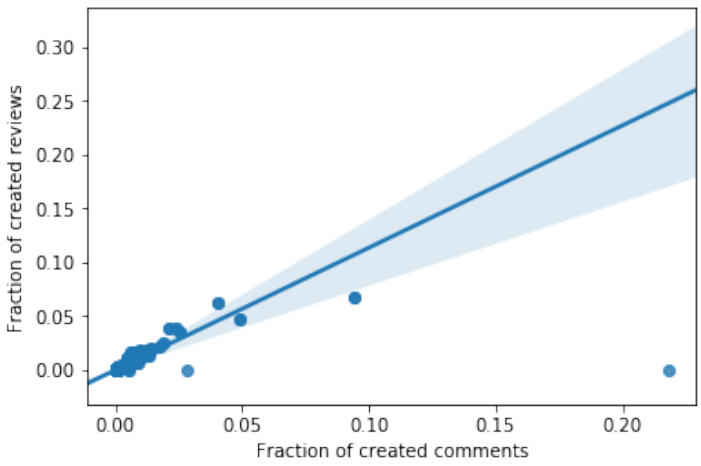

Figure 12: Fraction of created reviews vs. fraction of created comments in the Go project (regression excluding the outlier from the bot)

The outlier in the Figure 12, with a high fraction of comments, but no created reviews, is a bot in the Go project. Section 3.1.6 Descriptive Overview describes how bots were identified during the exploratory data analysis.
The bots in the different projects (except in the Typo3 project) have a strong impact on the number of created comments. The Table 6 shows the number of created comments by bots in the different projects.

	Bot ID	**Number of Created Comments**	**Percentage of Created Comments**
Gromacs	1000119	32233	27.1 %
Go	5976	70098	21.8 %
Typo3	45883	44	0.01 %

Table 6: Overview of Created Comments by Bots

The selection of the top-k developers was based on a threshold of 80% of the respective scores. By applying a different threshold, the number of developers in the selection will change. The percentage level for the threshold can be increased to better filter the top-k developers in the different projects. An overview of tested thresholds, the number of resulting developers and the percentage of their impact on the review process are listed in Table 7.

		Number of top-k developers	**Percentage of top-k developers**	**Percentage Created Reviews**	**Percentage Created Comments**	**Percentage Created Commits**	**Percentage Approved Reviews**
Gromacs	80 %	12	7.69 %	85.6 %	87.3 %	84.9 %	73.5 %
	85 %	11	7.05 %	85.0 %	86.7 %	84.2 %	73.3 %
	90%	8	5.13 %	78.7 %	82.2 %	78.1 %	69.6 %
	95 %	2	1.28 %	23.5 %	46.0 %	24.3 %	34.3 %
	98 %	2	1.28 %	23.5 %	46.0 %	24.3 %	34.3 %
	99 %	0	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %
Go	80 %	100	7.29 %	73.8 %	80.5 %	75.9 %	82.9 %
	85 %	80	5.84 %	70.9 %	78.2 %	72.7 %	80.6 %
	90 %	56	4.08 %	65.8 %	74.5 %	67.7 %	78.0 %
	95 %	27	1.70 %	53.7 %	65.6 %	55.4 %	71.4 %
	98 %	8	0.58 %	24.1 %	45.1 %	26.1 %	50.1 %
	99 %	5	0.36 %	19.6 %	41.4 %	21.8 %	41.9 %
Typo3	80 %	79	3.17 %	78.9 %	74.3 %	82.5 %	83.0 %
	85 %	58	2.33 %	75.8 %	72.5 %	80.0 %	81.6 %
	90 %	38	1.52 %	70.3 %	68.7 %	75.1 %	79.5 %
	95 %	21	0.84 %	51.2 %	53.7 %	56.2 %	62.4 %
	98 %	8	0.32 %	33.9 %	39.5 %	40.5 %	46.3 %
	99 %	5	0.20 %	22.6 %	30.0 %	27.2 %	32.8 %

Table 7: Number of top-k developers and their respective impact (measured in created reviews, created comments, created commits and approved reviews) depending on different selection thresholds

By applying an increasing threshold on the score values, the number of selected top-k developers decreases (e.g. 100 (80%), 80 (85%), 56 (90%), 27 (95%), 8 (98%), 5 (99%) for the Go project). However, the impact of the decreasing group of top-k developers on the project decreases slower, the individual impact of the remaining top-k developers increases.

This effect is shown in Figure 13, where the number of top-k developers and the average impact per top-k developer is shown as a function of the applied threshold for the Go project.

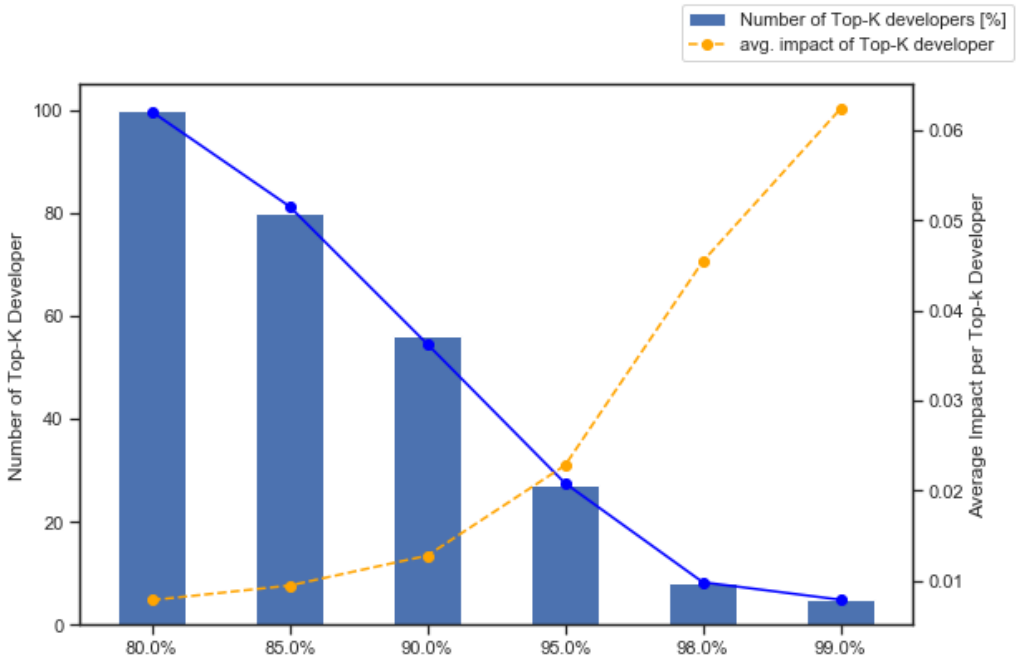

Figure 13: Number of top-k developers and the average impact per top-k developer as a function of the applied threshold for the Go project

2.7 Research Discussion

This thesis introduces several methods to identify active developers within the review process of different open source projects (i.e. Gromacs, Go and Typo3), both with statistical and with network theory measures. Resulting in multiple different approaches that allow identifying the top-k developers, which have a strong impact during the review process.

Seven different indicators to identify these developers were introduced and analyzed. All methods created almost equal selections of top-k developers; the respective scores showed a strong correlation.

Based on the different indicators, a method to combine these indicators in order to stabilize the selection process was presented and a final selection of top-k developers and their respective scores was calculated.

With this selection of top-k developers the general research question was answered by calculating their respective impact on the project.

The number of top-k developers depends on the applied threshold on the score values during the selection process. For example, with a 80% threshold, for the Go project, 100 developers (7.29% of all developers in the Go project) were selected as top-k developers having an about 80% impact (measured in created reviews, created comments, created commits and approved reviews) on the review process in the different projects.

Increasing the threshold for the selection reduces the number of selected top-k developers but the remaining developers have an increasing average impact per developer on the project (e.g. Go project: 80% threshold: 100 developers with 0.78% average impact; 99% threshold: 5 developers with 6.2% average impact)

These results show that the analyzed review processes of the open source projects (i.e. Gromacs, Go and Typo3) were dominated by a small number of very active developers (top-k developers). The impact of the top-k developers was measured based on the created reviews, created comments, created commits and approved reviews. In all these measures the selected group of top-k developers has a dominating impact.

Even for large open source projects as the Typo3 project with about 2500 contributors, a fraction of only about 1% of these contributors have a strong impact on the development, by approving and committing more than about 70% of the review content.

This analysis result implicates possible advantages, risks and introduces new questions for the future.

On the one hand, the small group of top-k developers are very important for the open source projects and take over key roles during the review process. They strongly participate in the project, by that they built-up expertise about the domain and the individual project and therefore they are crucial for the success of these projects.

On the other hand, the whole project is built-up around a core, a small group of developers, without these the outcome of the review process in the open source projects would be unpredictable and therefore this represents a possible risk.

As mentioned before, with the selection of the top-k developers, the project bot belongs to this group. The bot dominates the number of created comments, due to the fact that it provides feedback on the test suite results or from other automated processes. However, this fact does not reduce the impact of the human top-k developers. Beside its current function, the bot could be potentially used for additional purposes, as for example spreading knowledge within the project.

Gerrit has been developed since 2009 (see [3]), therefore the form of tooling provided by the Gerrit environment is still relatively new and only exists now for about ten years. As mentioned in the chapter 3.1.1 Code Review and Tool-Based Review, before review tools like Gerrit existed, there was rarely no data collection about the review process available or it was not easily accessible. The transition towards Gerrit, as review tool, took some time for the open source projects, as a consequence, there exists only a small collection of comparable data between different open source projects.

There exits research on the review process of open source projects in general but with no specific focus on identifying top-k developers within different projects.

In this thesis, it was shown, that there exists a small group of developers which represent the major participation in the review process of the projects: Gromacs, Go and Typo3. The methods introduced for identifying these top-k developers and the algorithms for preparing the Gerrit data can be used for further research.

Based on the analysis of top-k developers, the role of bots within the review process of open source projects may be expanded beyond their current function, e.g. using them for knowledge sharing.

All findings of this thesis are limited to the projects, the indicators and variables used to identify top-k developers and to analyze the impact of these developers.

For further research, this thesis suggests extending the analysis of the review process, in particular the impact of the top-k developers by analyzing measures like for example:

- Number of Insertions and Deletions of the reviewed code
- By selecting the developers that are able to approve/verify reviews
- Developers that disliked code
- Developers that verified code reviews
- Number of recommendation of reviewers

This thesis used data from the review process which was in conformity between the different projects. By focusing on individual differences between the different review processes used by the projects, impacts on the observed indicators may be analyzed to derive additional information and to investigate the impact on the projects.

Other network theory indicators for the community networks like cliques and maximal cliques in the networks could be tested as well.

To verify and to generalize the identification and the impact of the top-k developers, additional data sets like other open source projects mentioned in chapter 2.5.1 Data Source Origin or even completely different data sources like the commit histories in GitHub could be analyzed.

In addition to the generalization of the impact of top-k developers during the review process of open source projects, the impact on the open source project in general, of these top-k developers, could be analyzed as well.

Furthermore, the introduced methods and the resulting set of top-k developers could be used to study consequences (e.g. opportunities and risks) of this phenomenon on open source projects. What possibilities may arise from the knowledge of who these top-k developers are in a specific open source project.

A possible hypothesis could be, that large companies could use this information to manipulate, sabotage or even take over whole open source projects. In 2016 Jaruchotrattanasakul et al. [35] showed a method to use portals like GitHub for recruiting purpose. Therefore, it is possible that companies also use information from Gerrit to identify appropriate developers. Including additional information, as the names of the top-k developers of a given project, may open new possibilities in this field.

Reflecting this, another question could be, how open source projects should handle such critical information and what possible ways exist to avoid the risk of leaking this information.

Another possible research topic would be to test the positive or negative impact on the quality of the review process, by the fact that the process is dominated by a small group of top-k developers. Other studies like McIntosh et al. [36] recently demonstrated, that there is a negative impact on the quality of software if there is a low review participation. Based on the fact, that there are mainly the top-k developers reviewing and approving code, this could be an evidence that this negative impact also applies to open source projects.
As mentioned in the section 3.1.6 Descriptive Overview there are also many other indications that the review process is not very efficient, and it has a potential to improve.

2.8 Conclusion

In this thesis, code review data from Gerrit was analyzed. During the analysis, the hypothesis emerged, that there is a small collection of developers with a high impact on the review process in the open source projects. To further observe this phenomenon, three open source projects (i.e. Gromacs, Go and Typo3) were selected. It was observed that these three projects have a selection of active top-k developers, which have a high impact on the review process. The indicators used in this thesis, to identify the top-k developers in the review process, and the method presented to generate a final selection of developers showed that the identified top-k developers have a high individual impact on the review process.

Based on the fact that the results of this thesis only apply to the analyzed projects, further research could verify this approach and could generate a generalized method. The phenomenon observed in this thesis may be the tip on the iceberg. Succeeding research may design their research focus on the findings presented within this thesis.

3 Elaboration Chapter

3.1 Theoretical Foundations

3.1.1 Code Review and Tool-Based Review

Code Review is a fundamental part of the development process of software products (see [11]). This step is done by one or multiple persons to achieve the following goals:

- To identify defects and improve code quality (quality assurance activity).
- Learn from others and transfer knowledge.
- In some cases, to ensure compliance with QA guidelines and standards.

The persons performing the review are called reviewers. Code review is not only an important step in proprietary software development, but also in open source projects. In fact, it is best practice to perform a review for every code that gets included in the final software product (see [9]).

Basic code review methods comprise reading and viewing parts of the source code, which are affected by the change.

There are basically two types of code review (see [37]). First there is the pre-commit review concept. In this concept the code change is reviewed before it goes to the main repository. The second approach is post-commit review, that takes place after the code change has been included into the repository. Both concepts have their advantages and disadvantages. On the one hand, pre-commit slows down the integration of new code changes, but on the other hand, guarantees that only reviewed code gets integrated in the main repository and therefore stability and quality standards are ensured. Post-commit allows developers to commit changes continuously to the repository, by that increasing the speed of product changes, but increasing the risk that poor code makes it into the main repository.

For this thesis the pre-commit approach is relevant, because the used data set is based only on this concept.

Nowadays, the modern code review processes include additional automated tests, which provide a fast feedback about the applied code changes (e.g. syntax, build checks etc.).

This does not replace the review by a human being; but supports standardized minimum requirements on the review process. These controls are often combined with community style rules, that make the process of reading and combining code in projects simpler.

In the old days, reviewing code from open source projects was done by emailing code changes back and forth between developers (see [38]). With growth of open source projects and an increasing number of developers in the related community, this approach was not very efficient. Therefore, large open source projects use a tool-based review process, establishing a by far more productive and simpler communication and exchange during the review process.

Gerrit [3] is a free and web-based tool that supports the code review process and is used in many open source projects. It is based on the pre-commit review concept. Gerrit uses the git protocol [39] and therefore is directly connected to the used distributed version-control system (e.g. GitHub, Gitlab, etc.).

The general recommended workflow by Gerrit looks like this (see [40]):

1. Making the change
2. Creating the review
3. Reviewing the change
4. Reworking the change
5. Verifying and Submitting the change

The detailed review workflow differs from project to project, depending on community rules and settings on how to approve and verify changes.

The following example describes the workflow for the Go project (see [41]).

To modify the source code, the developer has to clone the project with the corresponding git command. To finish his change, he has to create a commit and connect it to a Change-Id. To finalize the commit, the author of the code modification can upload the commit as a change to Gerrit.

As next step, Go maintainers will do an initial reading of the change, followed by triggering "trybots". Bots that automatically check the full test suite of the Go project. The author of the review will be informed by notifications via email.

When the change is approved, that means, a review gets at least one +2[1] voting, an approver will apply and submit the change to the code base.

During the reviewing process there are many options to give feedback on the change, the author of the review has to answer every feedback and if needed has to improve the code by uploading the commit as a new patch set. Following this way, the change is improved iteratively and is only applied to the code base when it is finally accepted by the Go project community.

Normally, before a change gets included into the code base, the change not only needs to be approved by the reviewers but also needs to be verified. The verification step ensures, that the main code still compiles with the added change. As mentioned before, a trybot will be triggered for this purpose, this bot checks automatically if the code can compile.

The analysis in this thesis is based on the general review process and the respective procedures described above. However, there are slight differences between the open source projects and their use of the Gerrit tool. These differences are considered as non-relevant for the main statements of this analysis, as they are taken into account during the respective analysis steps.

3.1.2 Network Theory

This thesis uses Network Theory to identify top-k developers in the network structure of communication networks (see 2.5.2 Preparation for Network Analysis). To get the same theoretical foundation and wording when speaking of networks in this thesis, this section summarizes the main backgrounds.

First of all, within this thesis the terms graph and network are distinguished (see [29]). The graph term is used to describe the abstract representation of a network. While the network

[1] +2: The change is approved for being merged. Only Go maintainers can cast a +2 vote.

term combines the graph representation with additional information about the entities and relationships. Therefore, the structure of the communication networks in section 2.5.2 are graphs, but the additional information about the user nodes and the communication edges turn the abstract representation into a network.

A Graph, $G = (V, E)$ is a pair of a set of nodes (also known as vertexes) and a collection of edges (see [42]). An edge (u, v) is a pair of nodes. The Graph can either be undirected or directed. If a graph is directed, the order of the edge pair matters $(u, v) \neq (v, u)$.

For example, there are two nodes u and v, with an edge (u, v) between them. If the graph is undirected, the pair (u, v) is equal to (v, u). Else, if the graph is directed, the order within the edge matters and therefore $(u, v) \neq (v, u)$.

In general, a graph can have multiple edges between each pair of nodes.

Within this thesis, the python library networkx (see [2]) is used. This library provides the following four different graph types:

- Graph: undirected and no multiple edges
- DiGraph: directed and no multiple edges
- MultiGraph: undirected and multiple edges
- MultiDiGraph: directed and multiple edges

The Figure 14 displays the different types of graphs.

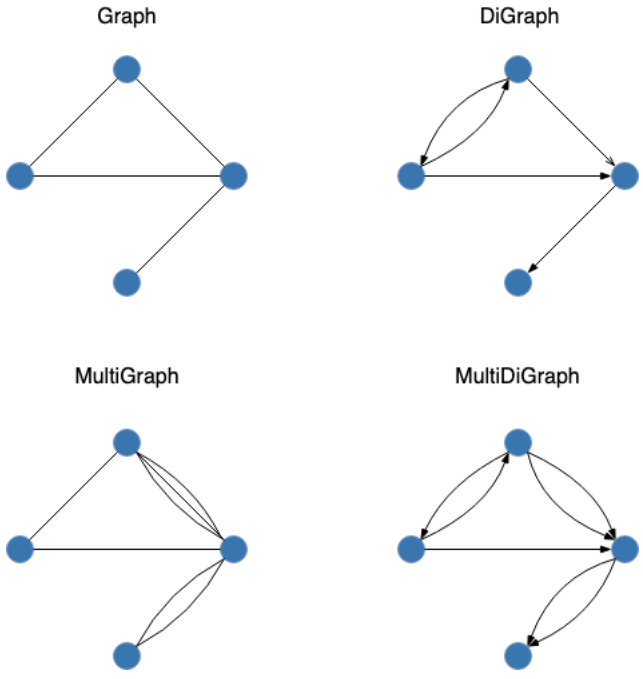

Figure 14: Overview of different Graph Types

Depending on the used graph model, the python library networkx needs to store a graph differently and only supplies a selection of possible algorithms for calculation of network indicators. Therefore, it is recommended to define the model as Graph or DiGraph, in order to ensure that these indicators are well defined.

The basic indicators which can be derived directly from the graph definition describe the size and the complexity of the graph and therefore directly characterize the network (see [29]).

- Number of nodes $n = |V|$
- Number of edges $m = |E|$
- Density d

The density of a network is a measure, that compares the amount of actual connections to the amount of potential connections. By that, it gives a first indication on the probability that two individual nodes in the network are potentially connected.

The density value is calculated differently if the graph is directed or undirected.

- $d = d_{directed} = \frac{m}{n\,(n-1)}$
- $d = d_{undirected} = \frac{2m}{n\,(n-1)}$

The numerical value of the density measure is in the range of zero to one for non-multigraphs.

A walk in a graph, is a series of edges $(u_1, v_1)(u_2, v_2)$ ... (u_p, v_p) with the length of p (see [42]). For $v_p = u_1$ the walk is closed. If all the edges of the walk are distinct, the walk is a trail. A trail where every u_i is distinct is called a path. And if the path is closed, it is called a cycle.

Another basic measure is the degree $\deg(v)$ of a node v. This value is the number of adjacent edges to node v.

The average degree in a graph is the sum of all node degrees, normalized by the number of nodes, it is calculated by:

$$\frac{2m}{n}$$

This measure gives a basic understanding of the average amount of edges connected to each node.

A more detailed introduction on Network Theory can be found in Estrada [42] and Zweig [29].

Descriptive Overview of the Available Data Set

3.1.3 Data Cleaning and Preparation

It is the goal of this chapter to generate a general overview of the provided data set and to create a base for further analysis and comparison of the different open source project instances. As first step of this process, an analysis of the content of the individual JSON review files was performed, followed by a cleaning of the data set in order to create a comparable base.

To work with JSON files in python, the content of the file gets converted into a JSON object. This object is comparable with a python dictionary. It uses structured key value pairs to describe the data that is stored within the object.

To compare the provided data from the different open source projects, an overview of all available key values for each project is generated. With the help of the function "JSON_flatten" from the library of the same name (see [43]), this overview was generated. The function reduces the hierarchy of the JSON object and therefore gives a suitable overview of all possible keys.

The generated overview revealed, that the projects differ in their JSON keys, which does not allow a simple comparison of the different open source projects. Therefore, a selection of those keys was chosen, that exists in at least ten out of all available twelve projects. In addition to this selection, all those keys, that do not provide any valuable content for this thesis, like for instance the key that stores the link to the user icon image, were removed from the selection.

Based on the fact that the extended thesis goal focused on three selected projects (Gromacs, Go and Typo3), the following chapters will exemplary focus on this selection as well.

Even if these projects were not the only projects that were analyzed in detail, this selection was verified to be suitable and provides a good base for the analysis.

The selection of the keys used in this thesis is presented in Figure 15, the color coding indicates the hierarchy of the JSON file (the top-level structure of the JSON file is described by different colors).

Keys	go	gromacs	typo3
_number	X	X	X
branch	X	X	X
change_id	X	X	X
created	X	X	X
current_revision	X	X	X
deletions	X	X	X
id	X	X	X
insertions	X	X	X
labels_Code-Review	X	X	X
labels_Verified		X	X
mergeable	X	X	X
messages__revision_number	X	X	X
messages_author	X	X	X
messages_date	X	X	X
messages_id	X	X	X
messages_message	X	X	X
messages_tag	X	X	X
owner_account_id	X	X	X
owner_email	X	X	X
problems_message	X	X	X
project	X	X	X
reviewers_REVIEWER	X	X	X
revisions_commit	X	X	X
revisions_created	X	X	X
revisions_files	X	X	X
revisions_kind	X	X	X
revisions_number	X	X	X
revisions_ref	X	X	X
revisions_uploader	X	X	X
status	X	X	X
subject	X	X	X
submit_type	X	X	X
submitted	X	X	X
topic	X	X	X
updated	X	X	X

Review
Revisions
Labels
Message

Figure 15: Selection of used JSON keys

After the selection of the JSON keys for this analysis, as next step, the data types and content of the various value fields are described. This provides an understanding of the data content.

Each individual review has multiple values that identifies itself.

- id — string value that uniquely identifies the review
- project — subproject name of the open source project
- branch — branch name the code of the review is located
- _number — number of the review, additional simpler identifier

The actual review data describes all further information about the review.

- change_id — id of the change number
- subject — subject title of the review, shortly describes the content of the review
- topic — keyword that can describe the content of the review
- status — current status of the review (ABANDONED, MERGED or NEW)
- created — date that the review was created
- updated — date that the review was last changed
- insertions — number of inserted and modified code lines
- deletions — number of deleted code lines
- mergable — status if the review code is mergable (true, false or empty)
- owner — user data set, who created the review
- submit_type — status of submit type
- submitted — date review got submitted
- labels — all possible values to evaluate the review and the actual review ratings, both for verification and code review
- reviewers — list of user data sets, who are the reviewer of this review
- messages — system and user messages about the review and information about the inline comments of the reviewed code
- current revision — current revision number of the review
- revisions — list of all review revision sets
- problems — list of system problems that occurred during the review

There are mainly four different data types used to store the value content: strings, dates, numbers and JSON structures. The sub JSON structures also store only these data types.

The shown overview of the selected JSON keys, their content and the used data types provide an understanding of the available data to be used within this analysis, but there are still issues preventing the straight forward analysis of the data set.

First of all, to read data of a review, the respective JSON file needs to be opened and parsed into a matching JSON object. As there are more than thirty thousand review files within one project (e.g. Go project), the simple approach would be to slow to work with the full data set.

To solve this problem, the data has been reorganized. The reorganization of data is described in the section 3.1.4 Data Reorganization and Storage.

Secondly, even if the projects provide a certain JSON key, this does not guarantee that this key is available in every review file. Sometimes there is an empty value like an empty string or list, but often the JSON key field is just left out. The handling of missing data is described in the section 3.1.5 Handling Missing Data.

Finally, there are differences in the sub-levels of the JSON structure between the different projects. For example, the user structure of the Gromacs project stores an id, name, username and email address. The Go project does not use any usernames. For the analysis this difference is not a problem, because the aim of identifying the user only requires a user id. With this user id the creator of a review or the author of a message can be unambiguously be identified. The processing of these differences is described in section 3.1.5 Handling Missing Data.

3.1.4 Data Reorganization and Storage

As first step of the preparation of the selected data a reorganization has been performed. In order to reduce the access to the individual JSON files during the analysis the data was parsed into tables with the support of pandas Data Frames. These tables were stored as csv files for the further analysis steps.

The simplest way to load a JSON object into a table, is by parsing the JSON data to a single table. Unfortunately, this method has many disadvantages.

As mentioned before, each JSON file has multiple levels, for example on the highest level there is the key user. The user value contains another JSON structure with additional keys, for the id, name, username and email address of this user. If the JSON file gets stored in the table structure, a column value can still be a JSON object. So, to access for example the id of the review owner, the owner column has to be read and then parsed into another JSON object. Only then the id can be read from this object. This makes it laborious to read and work with the data set.

Another disadvantage is, that the same information is stored in multiple keys. For example, every time a user occurs in the data, the full user information set (id, name, username and email) is stored. Even if the id uniquely identifies the user. This unnecessary data redundancy produces an inefficient data structure.

To avoid these disadvantages, the data will be reorganized into multiple tables. With the goal, to create a simple structure, removing data redundancy and facilitate an efficient reading and processing of the data set.

The new data structure comprises five tables for each project.

- user table: unique set of all project users
- message table: all the messages of the reviews
- revision table: all the revisions of the reviews
- label table: the label of each review
- review table: all reviews

Each message, revision and label, in the respective table, refers to a review using its unique identifier. Whenever a user occurs in one of these tables it refers to a user in the user table, using its unique user id, only in two special cases the user is identified by the user name.

With this structure, tables can be joined quickly and access to data is efficient and fast. This results in the base structure showed in the UML diagram Figure 16.

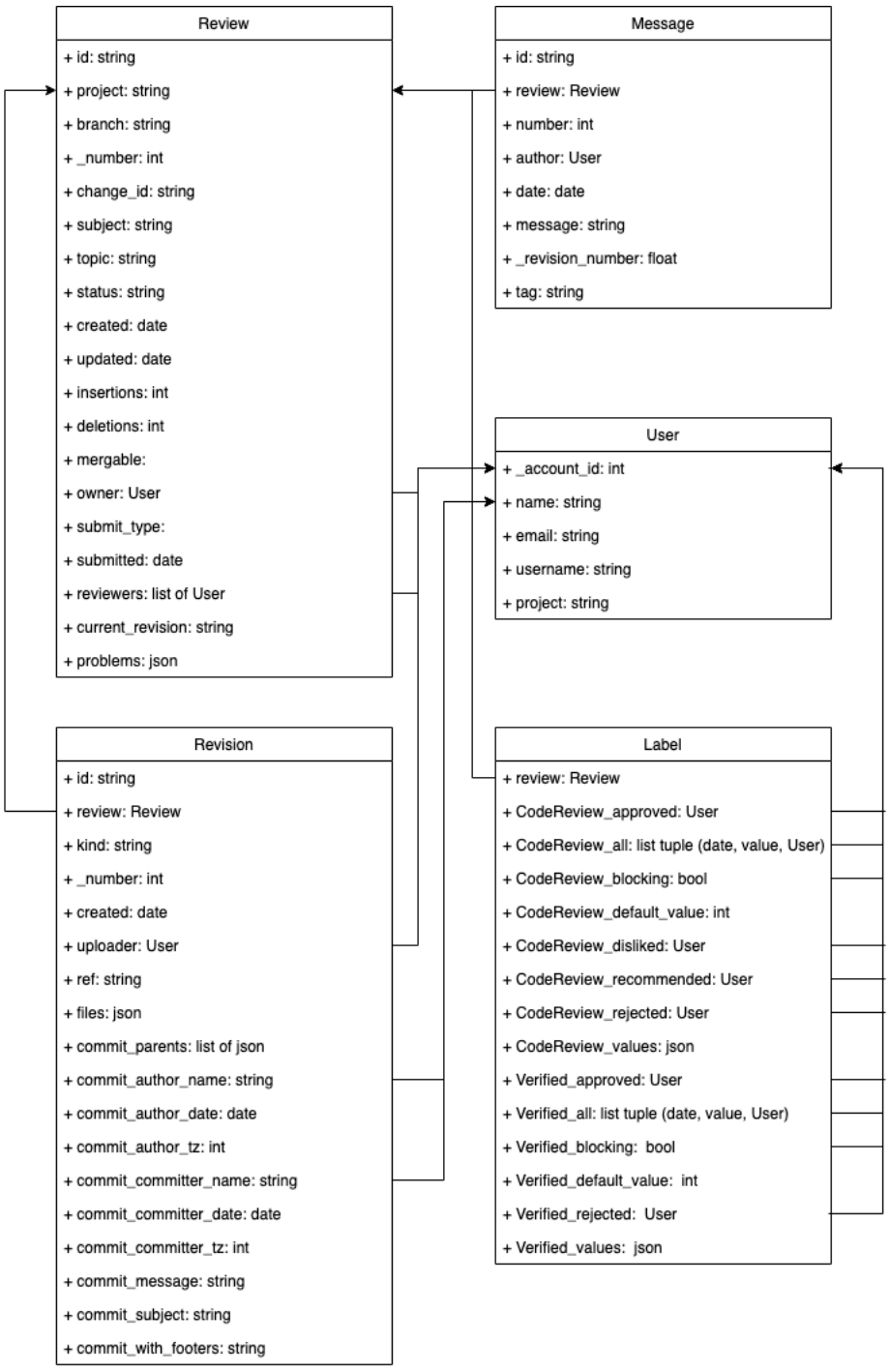

Figure 16: Structure of the storage and connection of the different CSV files

To fill these tables with data, every review file gets read and parsed into a JSON object. The associated data gets extracted from each JSON object and then gets inserted into the individual tables.

As mentioned before, only because a project has a certain JSON key, it does not mean that there is also a key value pair in every review. Whenever a value is missing an empty value (numpy NaN) will be inserted. This allows in a following process step to control the missing data and to check the reliability of each attribute stored in the tables. There will be a closer look at the handling of missing values in the following chapter 3.1.5 .

The last step of the data reorganization is to store these filled tables, in order to allow a later processing of this data. The advantage of the reorganization and storage of the newly generated tables is a fast access to the data, as it is not required to extract the data from the original JSON files every time it is used.

In addition, the described approach delivers the data already in pandas Data Frame format. Pandas comes with a set of options to store Data Frames, which make the storage convenient.

The csv format was chosen, because it allows quick access to the data set, offering comparatively small files and delivering files that are still readable and manipulatable with all common editors. Other formats like pickle, hdf or others were also considered but had slight disadvantages. For example, pickle “is not secure against erroneous or maliciously constructed data” [44], this makes pickle a format that is untrusted from unauthenticated sources (see [44]). Making it hard to publish or to hand-over the data to third parties. Other formats like hdf and hdf5 have the disadvantage that the data is stored in an own storage format, making the user depending on a specific parser.

The only disadvantage of storing the tables in csv format is, that when reading the files, the data types of each attribute in the table gets predicted. This makes it complicated to store data types like python lists. This disadvantage is acceptable, because there is only one case in which python lists were used. For this case, an additional parsing step is required.

The csv files are provided in the following structure. There is a data folder, with three individual sub-folders for Gromacs, Typo3 and Go project data. In each project folder there are the five csv files storing all the selected review data. Each file is named by the name of the project and the content it stores. The csv files use semicolons as delimiter.

3.1.5 Handling Missing Data

The previous sections described how the data has been prepared, structured and stored. The data can be loaded quickly to perform further analysis. Previously left out values in the json files got replaced with numpy NaN values. On the one hand, this is very useful because an overview of the amount of left out values can be generated. On the other hand, while working with the data, attributes in the table can contain empty (numpy NaN) values. So, whenever an attribute value gets read or compared with another value, there must be a check for empty values before. To avoid this problem, this chapter examines NaN values, checks how often these values occur, and investigates whether they are a problem. The chapter finally presents a solution to deal with the missing data and finish the data preparation.

To count how often empty (numpy NaN) values occur for each stored attribute, the table data gets loaded into a pandas DataFrame format.

For each column the number of occurring numpy NaN values gets calculated. This method generates a full overview about the number and percentage of NaN values in the different tables.

A first observation showed that some provided functions of the Gerrit review process are not used by all projects and that there are some fields which contain unexpectedly empty values.

Depending on the project, there are different information missing. In the following, every abnormality will be looked at and discussed.

First of all, there are cases where it is expected that the respective json files do not contain values, because not every review uses this information. For example, not every review gets submitted, therefore has a submitted date and not every review needs to have a topic. These cases are plausible, more interesting are the cases where values are unexpected missing. Secondly there are some fields that are completely missing for one project, this is also not unexpected because each key was chosen when at least ten of the investigated twelve projects had this information. So, this information can be missing for a maximum of two projects.

In the following there will be a closer look at those cases, where the missing data is not expected.

In the Gromacs project, there is one review without the insertions and deletions key, unfortunately no clear reason for this occurrence could be identified. As this exception only appeared once in 7457 reviews; this exception was ignored.

72 reviews had no reviewers tag, even though in most of these cases the review was abandoned or still new, there were still three merged reviews without reviewers. These three reviews were in two different branches and had normal changes on the code base, they showed no messages or signs of other users except from the owner, in other respects the reviews are normal. Again, there was no real reason identifiable for this occurrence and therefore it was ignored.

Besides that, not every user in the Gromacs data had a name, email or username. It turned out that Gromacs has, in contrast to the Go project, the feature to mark accounts as deleted and therefore loses some of this information. Fortunately, this additional information is almost irrelevant, because the system still stores the account ids, so there is no problem identifying connections.

Bot accounts do not get deleted, so there is no problem to identify bots by their name.

The last issue with the project was, that 2421 messages had no author. This is about 32% of all the messages in the Gromacs project. It turned out, that these messages were system messages, that are generated automatically and therefore do not have an author.

The other two projects (Typo3 and Go) displayed the same findings as the Gromacs project. Due to the fact, that these two projects contain more collected reviews in their data, the effects like the number of missing insertions and deletions information were slightly higher. Without loss of generality these cases were, as in the Gromacs project, ignored.

A difference of the Go project was a lack of usernames, the Go project does not use usernames but user ids. This is no problem for further analysis, but again shows that even if all these open source projects use Gerrit as review tool, they use it in their own specific way.

The analysis of the missing data provided further understanding of the way the Gerrit tool works and how each of the analyzed projects defines its review process.

There were no reasons found which would prevent using the data for the analysis purpose of this thesis.

To finalize the missing data handling, the remaining numpy NaN will be replaced by a default value. This allows to analyze the data without checking every value for an empty value in each circumstance.

In columns that store numbers, the NaN value gets replaced by values that describe the default case. Like whenever the number of insertions or deletions is missing, this NaN value will be replaced by zero. Because looking at the review there are no insertions or deletions.

But there always will be values that do not need any and should not have a default value, like the id of a review. This value should never be missing.

The matching default values for each attribute in all the tables are listed in the Appendix D.

Every numpy NaN value gets replaced by the corresponding default value from this figure.

This results in the final base of the whole data set. The base is now complete, structured and cleaned.

3.1.6 Descriptive Overview

After the preparation and cleaning steps, an exploratory data analysis was executed. This was the original thesis topic and for that all available values in the prepared tables were considered and analyzed. The collected review data of the different open source projects reflects the project, the development process and the persons engaged and involved within the projects. It describes the open source projects and therefore gives insights into internal processes.

This section will give a descriptive overview of the review data by focusing on selected phenomena. By that, it provides additional insights in comparison to the analysis presented in the research chapter.

Work during Working Time

As described in the theoretical chapter, every code artifact written for an open source project will be reviewed and the code review is a mandatory step during the process of developing and participating in an open source project. It is actual one of the most important tasks during the software development process.

It can be concluded that a developer which participates in the review process does participate in the open source project and the time spent in the review process is an indicator for the participation in the project.

In other works, like Riehle et al. [45], it has been shown that the open source project developers participate not only during their free time but most often during normal working hours. Actually, it is normal that “about 50% of all work contributed to open source projects has been provided Monday to Friday, between 9am and 5pm” [45].

By analyzing the different time stamps during the review process, the working time period is also dominated in this data set. For example, most of the created and updated reviews were done from Monday to Friday, shown in Figure 17.

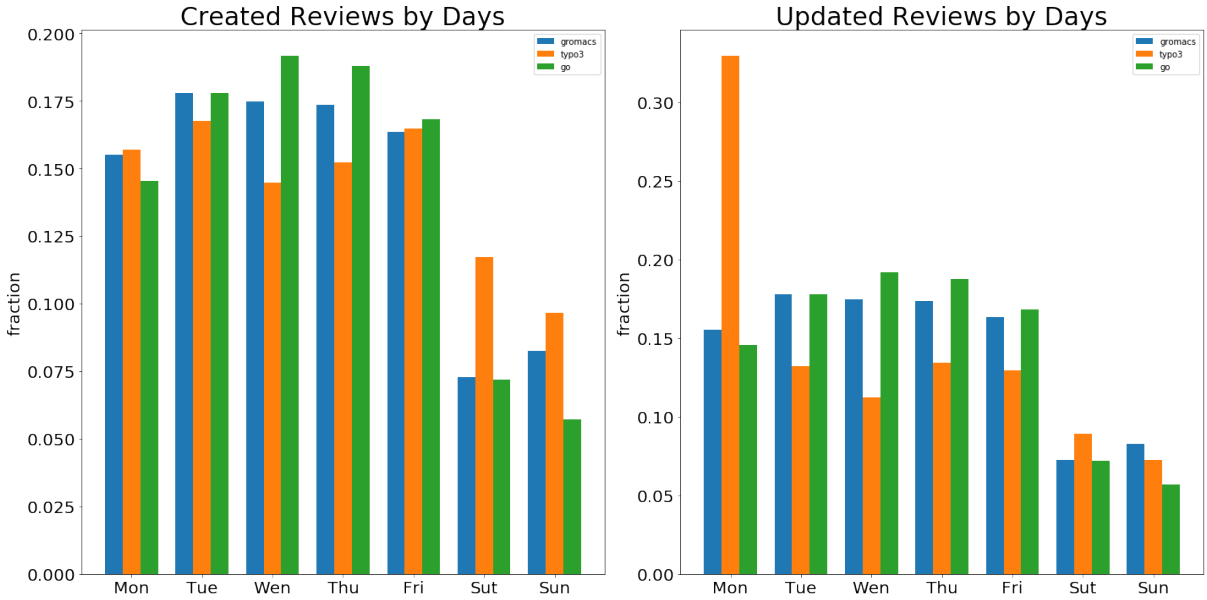

Figure 17: Fraction of Created and Updated Reviews for the open source projects (Gromacs, Go, Typo3) by Weekday

The actual work hours (derived from the time stamps) do not support this effect, the reason is most likely that the developers work in different time zones and Gerrit automatically generates time stamps based on the time of the server system. The work distribution on the weekdays indicate that the developers participating in the open source project most likely participate during the normal work day. In further research work, this effect could be further analyzed.

Identification of Bots

Another topic already mentioned in the chapter 3.1.1 are automated tests and feedbacks by triggered mechanisms. As the Go review workflow already describes, there is a so-called trigger bot (see chapter 3.1.1 Code Review and Tool-Based Review) that automatically tests the complete test suite. The exploratory data analysis emphasized that this bot has an account in the Gerrit environment to deliver direct feedback to the community. It turned out, that each project has their own bot system implemented. Therefore, this thesis decided to identify these bots to distinguish if the participation and impact during the review process was automatically generated or a real contribution.

After testing different approaches, it became obvious that the bots can simply be identified by their name. In all three projects analyzed in this thesis (Gromacs, Go and Typo3), the bot user had the keyword ‘Bot’ in his name. With this information, the bot user could be filtered out, resulting in the following list of bot users (see Table 8).

Project	Id	Name
Gromacs	1000119	Jenkins Buildbot
Go	5976	Gobot Gobot
	12446	Gerrit Bot
Typo3	45883	TYPO3 Bot

Table 8: Overview Bots in the different projects (i.e. Gromacs, Go, Typo3)

Productivity and good style of Code Development

As mentioned earlier (see 3.1.1 Code Review and Tool-Based Review) the code review task is used as quality assurance during the development process. In fact, code review attempts to improve the quality of the code base. In addition to that, there have been best practices (e.g. see [46] or [47]) to improve the code review and to increase the code quality during the development (see [11] and [48]).

For example, from the perspective of the review owner:

- Create small code changes (simple to read, best case incremental)
- Decide on the number of reviewers (e.g. a work at Microsoft [46] suggest about two reviewers)

From the perspective of the reviewer:

- Review frequently
- Take enough time to understand code (e.g. a work at IBM [47] suggests less than 400 lines of code)

With this selection of simple concepts in mind, there are many indications in the code review data set from Gerrit, that potentially show that the review process in the different open source projects (i.e. Gromacs, Go and Typo3) were not very effective.

First of all, the number of insertions and deletions, these numbers represent the size of the code change related to the review. The following boxplots in Figure 18 show the average insertion size of a code review created in Gerrit for the three different projects (i.e. Gromacs, Go and Typo3), excluding outliers. From this point of view, it seems that the size of the changes is actually in the range of an optimal suggested size, like explained above.

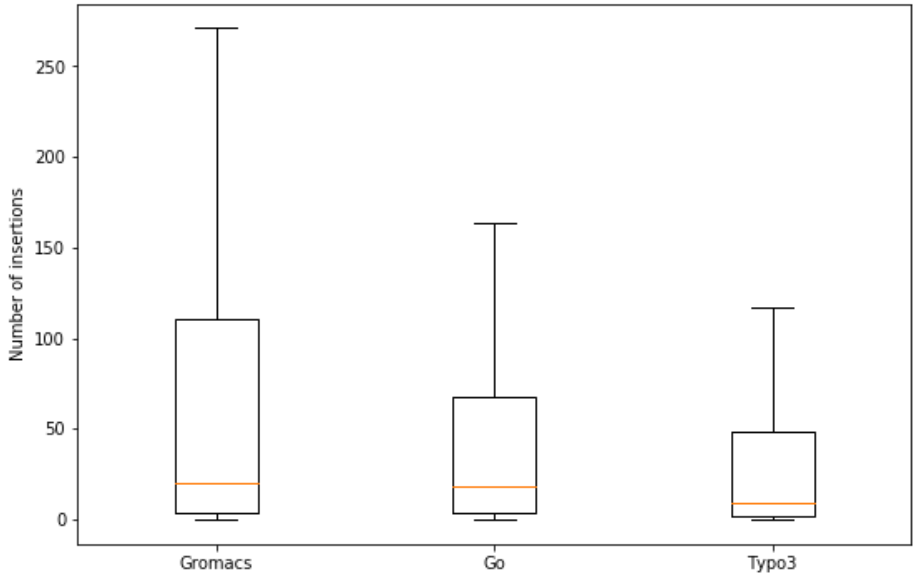

Figure 18: Boxplot of Insertions per Project (excluding outliers)

But on the other hand, there are 4656 reviews in the Typo3 project that are larger than the maximum of the boxplot in Figure 18 with numbers of up to 1,711,760 (insertions/deletions) lines of code. A similar effect could be seen in all three projects. These large changes are not only changes of the structure of the system, which would not add new functionalities to the code base, e.g. the movement of a development branch into the master branch. There are large changes that implement new features.

Another possible indicator is the time delta between the creation of a review and the last change on it. Figure 19 displays this time delta in days for the projects (i.e. Gromacs, Go and Typo3), excluding outliers.

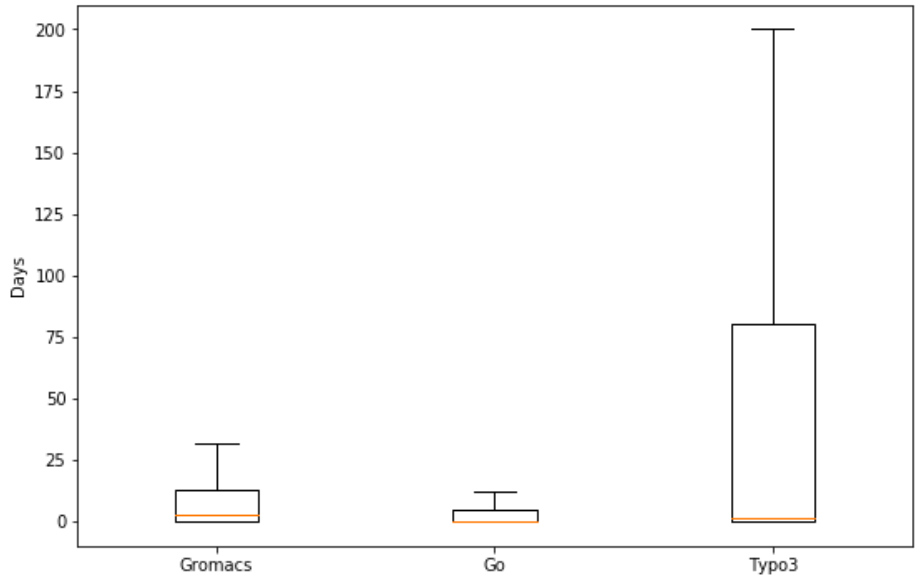

Figure 19: Boxpot of time delta between review creation and last change in days (excluding outliers)

Again, the outliers show the extremes of possible values, there are reviews that are open for more than 2000 days in the Typo3 project and the average time that a review is open goes up to 100 days in the Typo3 project including outliers. The other projects (i.e. Gromacs and Go) showed similar results. This could also be an indicator for a lack of productivity and a potential to improve.

Further research could elaborate on the productivity of these open source projects and analyze if they lack efficient quality assurance procedures. Additionally, the different projects could be compared against each other to establish best practices.

3.2 Used Tools and Libraries

During the process of creating this thesis python in the version '3.7.2' was used to prepare, to analyze and to work with the data (see [1]). As development environment the jupyter project was used (see [49]). This environment allowed an interactive interaction with the provided review data.

The python libarys:

- Pandas [50]
- Numpy [51]
- Matplotlib [52]
- Seaborn [53]

were used to support the complete interaction with the data set, to provide a productive working environment and to serve the fundaments for scientific computing with python.

For network analysis, the library networkx was used, except for the creation of the visualization of the graphs. Because networkx lacks visualization capabilities, python-igraph was used instead. This library is designed, optimized and dedicated to this task.

Additional libraries like tdqm (better progress bars in jupyter), json_flatten (see the descriptive overview chapter) and others were used to support the progress of this thesis.

Appendix A Identification Figures Typo3 and Gromacs

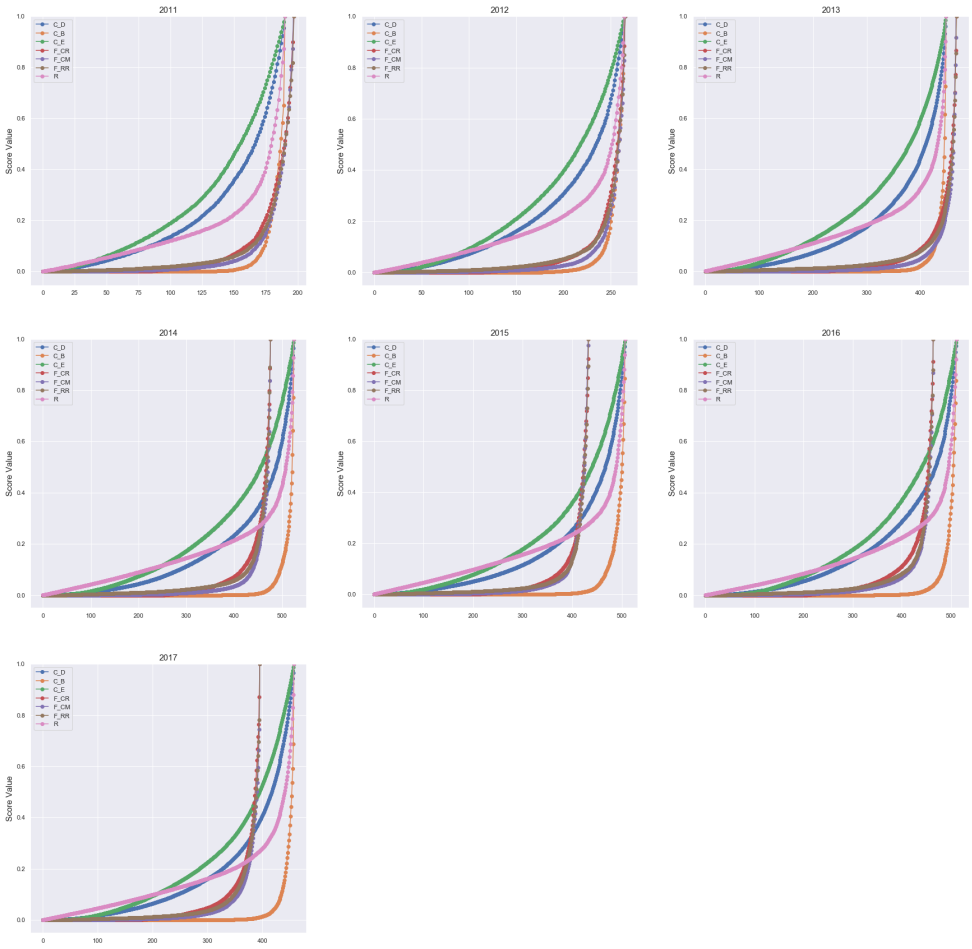

Figure 20: Cumulative sums of all seven indicators versus the sorted (by score contribution) active developers for the Typo3 project from 2011 to 2017

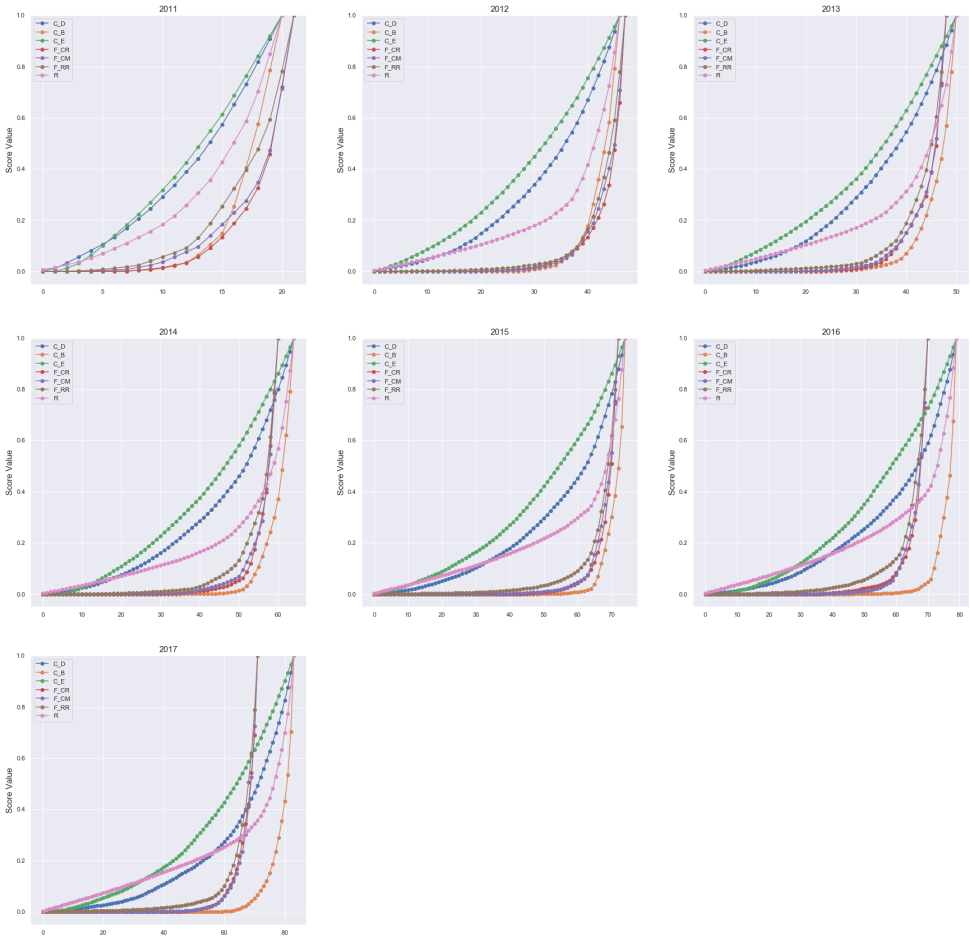

Figure 21: Cumulative sums of all seven indicators versus the sorted (by score contribution) active developers for the Gromacs project from 2011 to 2017

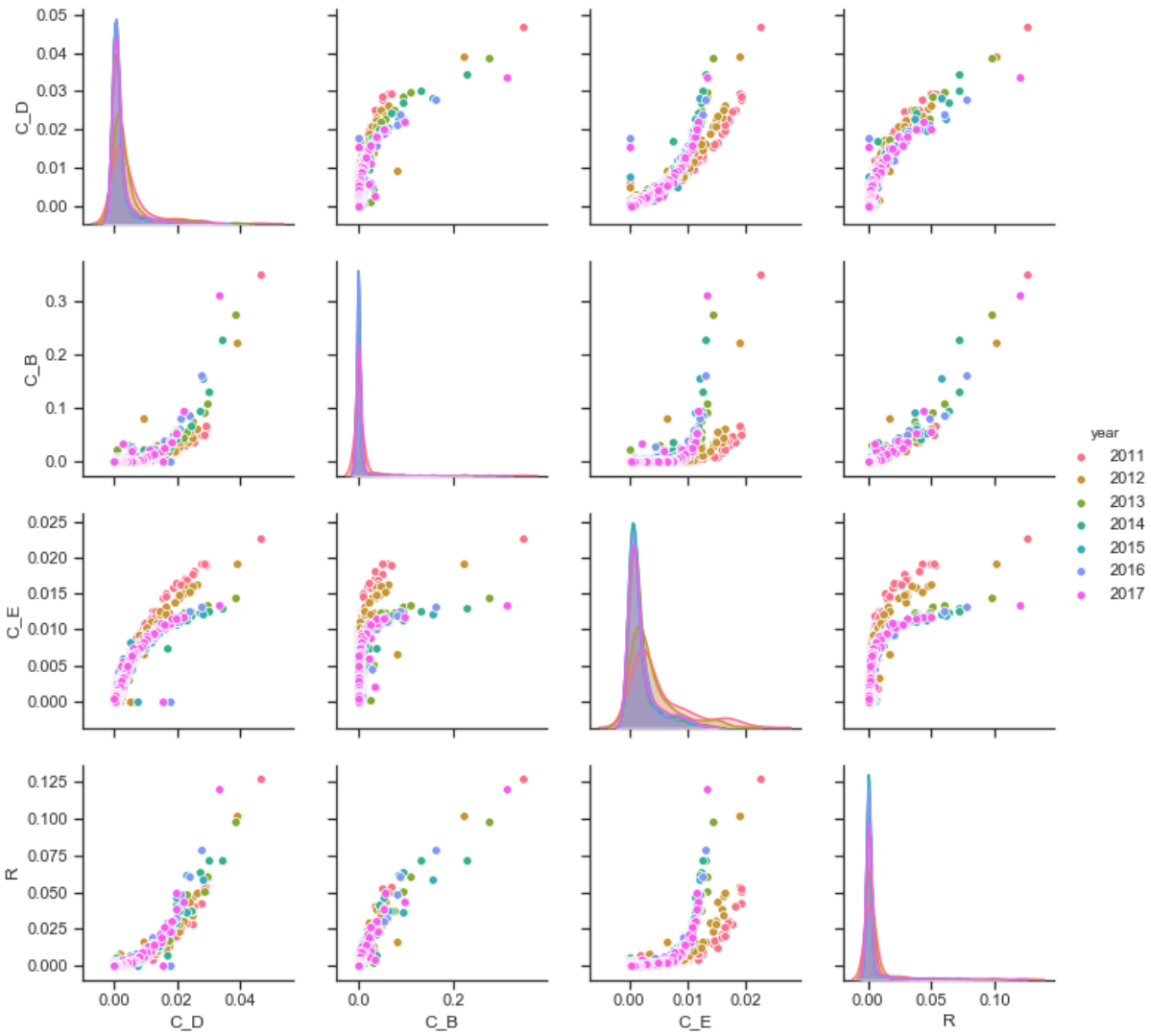

Figure 22: Pair Plot (multiple scatter plots) of the network theory indicators in the Typo3 project from 2011 to 2017

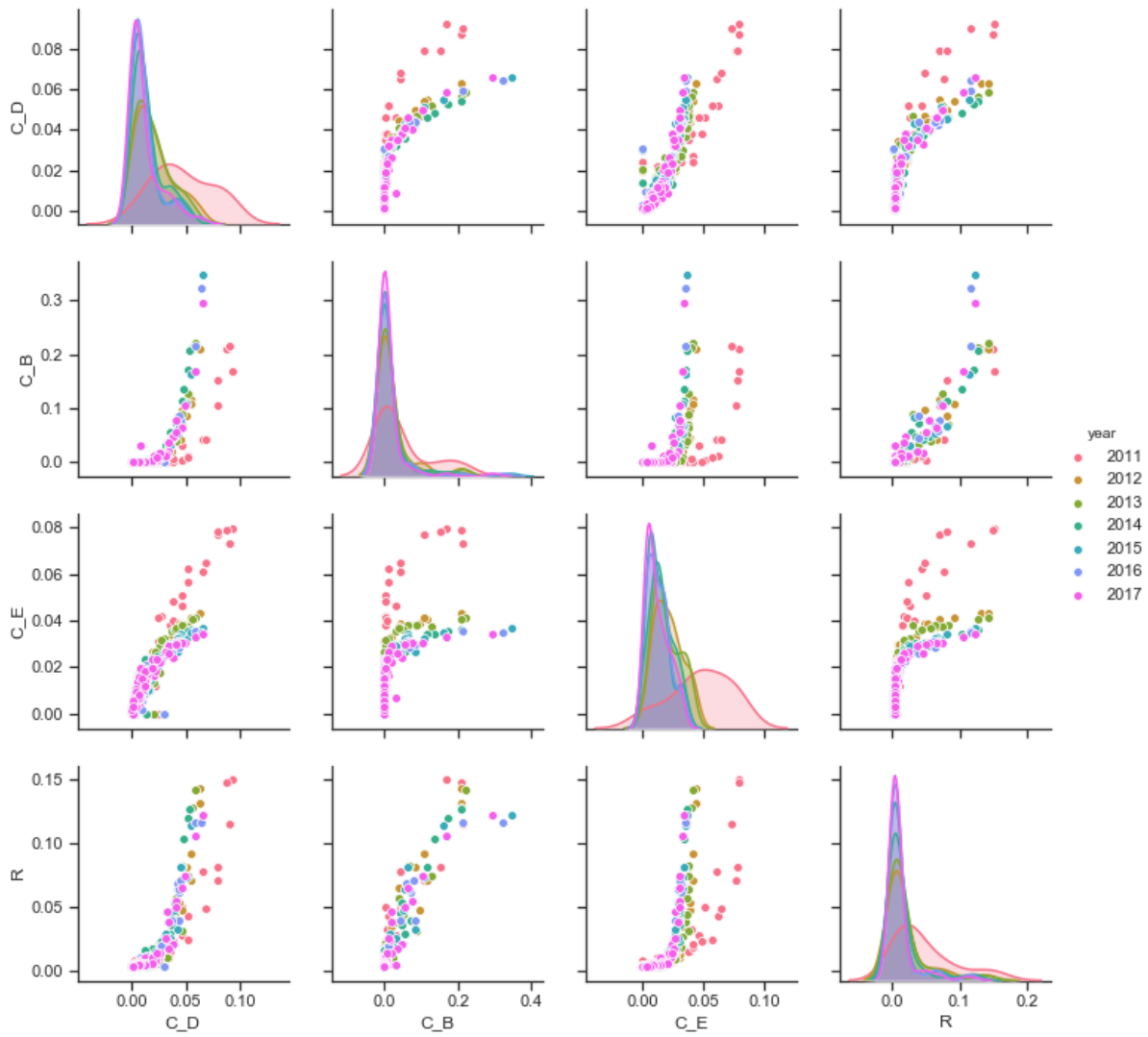

Figure 23: Pair Plot (multiple scatter plots) of the network theory indicators in the Gromacs project from 2011 to 2017

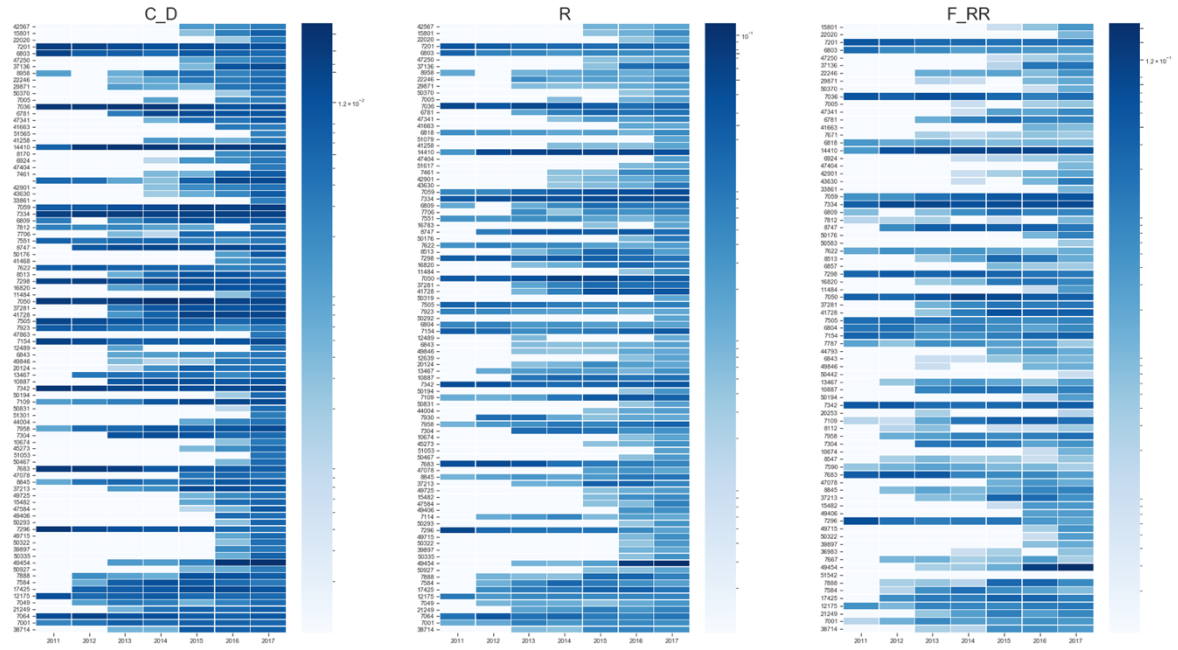

Figure 24: Top-k developers and their different indicator scores for the Typo3 project

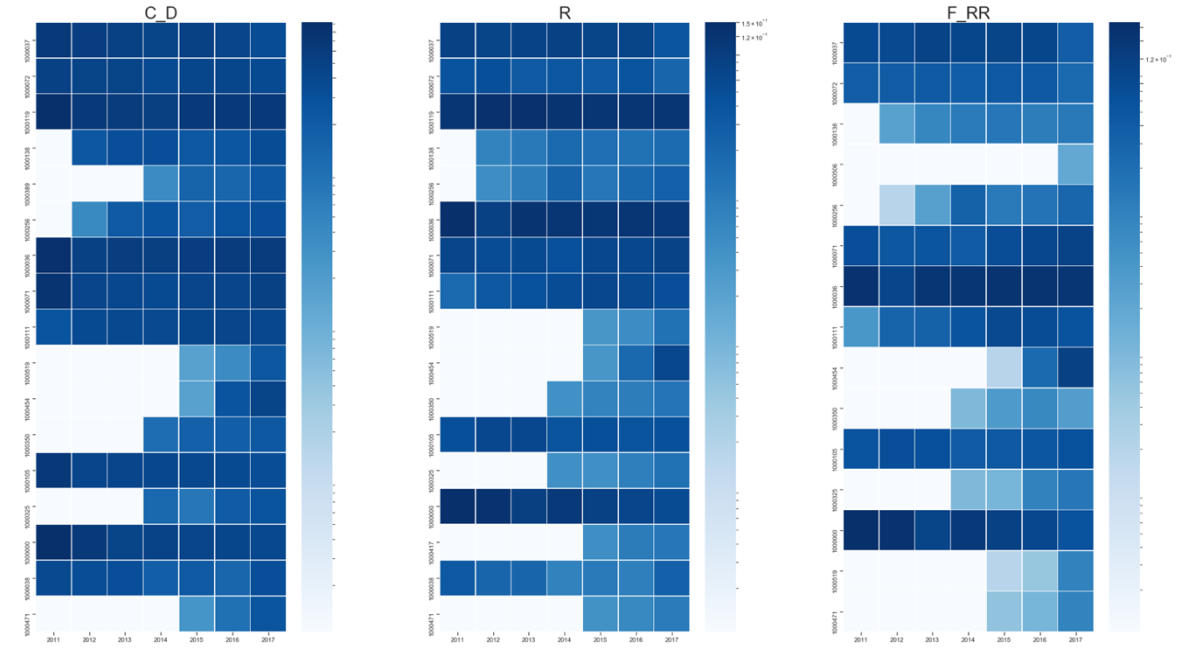

Figure 25: Top-k developers and their different indicator scores for the Gromacs project

Appendix B Final lists of top-k developers

39915	13832	7107	9308	7201	6803	7927	37220	8958
6917	6881	7036	6781	6818	21396	11141	14410	10279
7444	11433	7059	7334	14642	6809	8844	7551	7028
8747	6904	11089	7622	8513	7178	7298	10876	7050
37281	41728	7505	7923	7206	37830	7397	6804	7154
7787	6825	9250	6906	13467	10887	7342	9149	7383
7109	7930	8267	7958	7304	6763	7590	7838	7683
8845	12283	7114	7296	6928	6807	7584	17425	12175
7049	7075	21249	7899	7146	7064	7001		

Table 9: Typo 3 final list of top-k developers (Threshold 80%)

1000037	1000072	1000138	1000119*	1000108	1000256	1000111	1000036	1000071
1000454	1000105	1000000						

*Table 10: Gromacs final list of top-k developers (Threshold 80% and Bot marked *)*

Appendix C Impact of the top-k developers

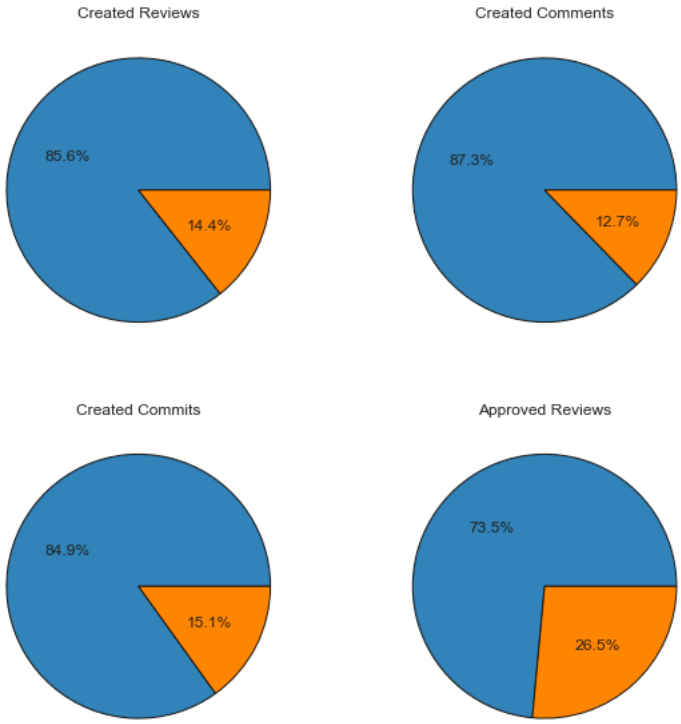

Figure 26: Impact of the top-k developers in the gromacs project (top-k developer contribution is marked blue;contribution of all developers is marked orange)

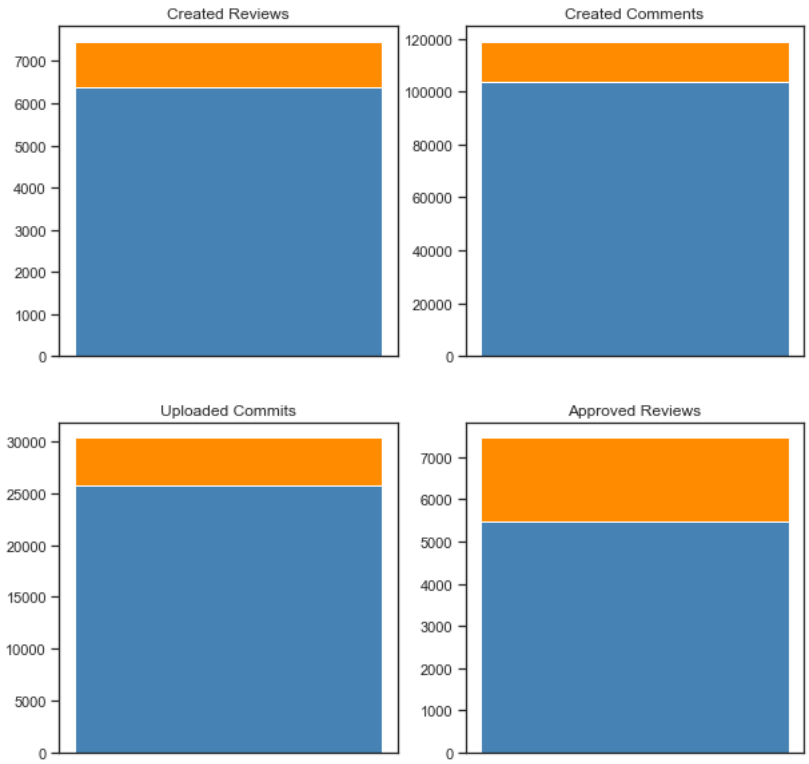

Figure 27: Impact of the top-k developers in the gromacs project (top-k developer contribution is marked blue; contribution of all developers is marked orange)

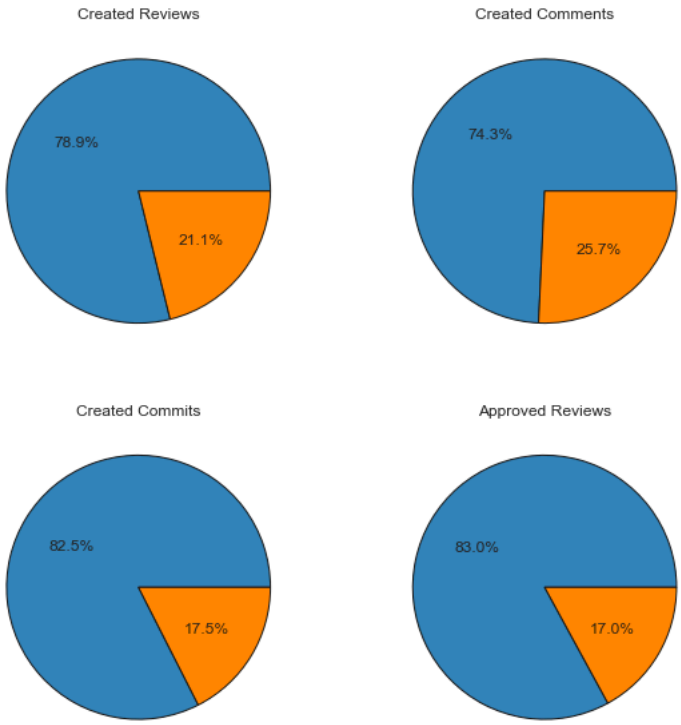

Figure 28: Impact of the top-k developers in the typo3 project (top-k developer contribution is marked blue;contribution of all developers is marked orange)

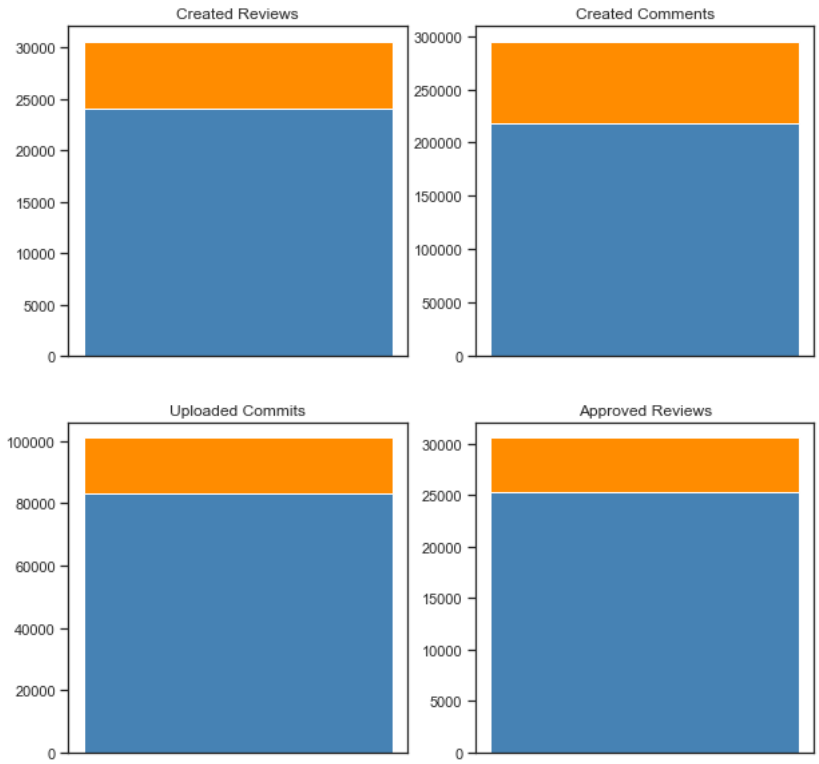

Figure 29: Impact of the top-k developers in the typo3 project (top-k developer contribution is marked blue; contribution of all developers is marked orange)

Appendix D Overview of default values

Column Name:	Default Value:	Description:
Table Reviews		
id / _number	-	Identifier should not be empty
project / branch	“”	
change_id	“”	
subject / topic	“”	
status	UNKNOWN	
created/updated	-	Automatically generated, should not be empty
insertions / deletions	0	reviews without these keys do not have any changes on the code, so zero rows are modified
mergable	UNKNOWN	
owner	-	Should not be empty
submit_type	“”	
submitted	“”	
reviewers	[]	empty list
current_revision	“”	
problems	[]	empty list
Table revisions:		
Id / _number & review	-	Should not be empty
Kind	UNKNOWN	
Created	-	Autmoatically generated
Uploader	-	Should not be empty
Ref	“”	
Files	{}	
Commit_parents	[]	
Commit_author_name / Commit_author_date / Commit_author_tz	“”	
Commit_committer_name / Commit_committer_date / Commit_committer_tz	“”	
Commit_message & Commit_subject	“”	
Commit_with_footers	“”	
Table labels:		
review	-	
CodeReview_approved /	“”	

Verified_approved		
CodeReview_all / Verified_all	[]	
CodeReview_blocking / Verified_blocking	UNKNOWN	
CodeReview_default_value / Verified_default_value	NaN	
CodeReview_disliked	{}	
CodeReview_recom-mended	{}	
CodeReview_rejected / Verified_rejected	{}	
CodeReview_values / Verified_values	{}	
Table user:		
_account_id	-	Should not be empty
Name / email /username	“”	
Table message:		
Id / number & review		
Author	“”	
date	“”	
message	“”	
_revision_number	NaN	
tag	“”	

Table 11: Overview default values in the different review data tables

References

[1] "Python." [Online]. Available: https://www.python.org/.
[2] "Networkx." [Online]. Available: https://networkx.github.io/.
[3] "Gerrit." [Online]. Available: https://www.gerritcodereview.com/.
[4] M. Mukadam, C. Bird, and P. C. Rigby, "Gerrit software code review data from Android," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013.
[5] R. Mishra and A. Sureka, "Mining Peer Code Review System for Computing Effort and Contribution Metrics for Patch Reviewers," in *2014 IEEE 4th Workshop on Mining Unstructured Data*, 2014.
[6] N. Kitagawa, H. Hata, A. Ihara, K. Kogiso, and K. Matsumoto, "Code review participation," in *Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering - CHASE \textquotesingle16*, 2016.
[7] J. Lipcak and B. Rossi, "A Large-Scale Study on Source Code Reviewer Recommendation," in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2018.
[8] D. Spadini, M. Aniche, M.-A. Storey, M. Bruntink, and A. Bacchelli, "When testing meets code review," in *Proceedings of the 40th International Conference on Software Engineering - ICSE \textquotesingle18*, 2018.
[9] P. C. Rigby, D. M. German, and M.-A. Storey, "Open source software peer review practices," in *Proceedings of the 13th international conference on Software engineering - ICSE \textquotesingle08*, 2008.
[10] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey, "Investigating code review quality: Do people and participation matter?," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015.
[11] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013.
[12] M. Sharma, U. Sharma, S. Jain, and S. K. Khatri, "Discovering Influential Nodes on Social Media using Gaussian Kernel Method," in *2019 Amity International Conference on Artificial Intelligence (AICAI)*, 2019.
[13] D. Kempe, J. Kleinberg, and É. Tardos, "Maximizing the spread of influence through a social network," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD \textquotesingle03*, 2003, pp. 137–146.
[14] J. Scripps, "Discovering Influential Nodes in Social Networks through Community Finding," in *Proceedings of the 9th International Conference on Web Information Systems and Technologies*, 2013.
[15] J. Shetty and J. Adibi, "Discovering important nodes through graph entropy the case of Enron email database," in *Proceedings of the 3rd international workshop on Link discovery - LinkKDD \textquotesingle05*, 2005.
[16] H. Mahyar, "Detection of Top-K Central Nodes in Social Networks," in *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2015 - ASONAM \textquotesingle15*, 2015.
[17] R. Nema and A. Pandey, "Community Kernels Detection in OSN using SVM Clustering and Classification," *Int. J. Comput. Appl.*, vol. 113, no. 11, pp. 9–13, Mar. 2015.
[18] S. Wen, J. Jiang, Y. Xiang, S. Yu, and W. Zhou, "Are the popular users always important for information dissemination in online social networks?," *IEEE Netw.*, vol. 28, no. 5, pp. 64–67, Sep. 2014.
[19] X. Zhao, F. Liu, J. Wang, and T. Li, "Evaluating Influential Nodes in Social Networks by Local Centrality with a Coefficient," *ISPRS Int. J. Geo-Inf.*, vol. 6, no. 2, p. 35, Jan. 2017.
[20] S. Huang, T. Lv, X. Zhang, Y. Yang, W. Zheng, and C. Wen, "Identifying Node Role in Social Network Based on Multiple Indicators," *PLoS ONE*, vol. 9, no. 8, p. e103733, Aug.

2014.
[21] G. Ghoshal and A.-L. Barabási, “Ranking stability and super-stable nodes in complex networks,” *Nat. Commun.*, vol. 2, no. 1, Jul. 2011.
[22] J. W. Tukey, *Exploratory data analysis*. Addison-Wesley, 1977.
[23] K.-J. Stol and B. Fitzgerald, “The ABC of Software Engineering Research,” *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 3, pp. 11–12, Sep. 2018.
[24] “Gerry Crawler GitHub - https://github.com/michaeldorner/gerry.” .
[25] H. D. Michael Dorner Maximilian Capraro, Dirk Riehle, “Gerry – A Crawler for Avoiding Threats to Gerrit Review Data Integrity,” Friedrich-Alexander-University Erlangen-Nuernberg, unpublished (privately communicated).
[26] “Graphml.” [Online]. Available: http://graphml.graphdrawing.org/.
[27] “Igraph Python.” [Online]. Available: https://igraph.org/python/.
[28] T. M. J. Fruchterman and E. M. Reingold, “Graph drawing by force-directed placement,” *Softw. Pract. Exp.*, vol. 21, no. 11, pp. 1129–1164, Nov. 1991.
[29] K. A. Zweig, *Network Analysis Literacy*. Springer-Verlag KG, 2015.
[30] S. P. Borgatti and M. G. Everett, “A Graph-theoretic perspective on centrality,” *Soc. Netw.*, vol. 28, no. 4, pp. 466–484, Oct. 2006.
[31] U. Brandes, “A faster algorithm for betweenness centrality\ast,” *J. Math. Sociol.*, vol. 25, no. 2, pp. 163–177, Jun. 2001.
[32] “Networkx Betweenness Centrality.” [Online]. Available: https://networkx.github.io/documentation/networkx-1.10/reference/generated/networkx.algorithms.centrality.betweenness_centrality.html.
[33] M. E. J. Newman, *Networks: An Introduction*. Oxford University Press, 2010.
[34] L. Page, S. Brin, R. Motwani, and T. Winograd, “The PageRank Citation Ranking: Bringing Order to the Web.,” Stanford InfoLab, Technical Report 1999–66, Nov. 1999.
[35] T. Jaruchotrattanasakul, X. Yang, E. Makihara, K. Fujiwara, and H. Iida, “Open Source Resume (OSR): A Visualization Tool for Presenting OSS Biographies of Developers,” in *2016 7th International Workshop on Empirical Software Engineering in Practice (IWESEP)*, 2016.
[36] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, “The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the Qt, VTK, and ITK Projects,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, New York, NY, USA, 2014, pp. 192–201.
[37] T. Baum, F. Kortum, K. Schneider, A. Brack, and J. Schauder, “Comparing pre-commit reviews and post-commit reviews using process simulation,” *J. Softw. Evol. Process*, vol. 29, no. 11, p. e1865, Apr. 2017.
[38] P. C. Rigby, D. M. German, L. Cowen, and M.-A. Storey, “Peer Review on Open-Source Software Projects: Parameters, Statistical Models, and Theory,” *ACM Trans Softw Eng Methodol*, vol. 23, no. 4, pp. 35:1–35:33, Sep. 2014.
[39] “Git.” [Online]. Available: https://git-scm.com/.
[40] “Gerrit Workflow.” [Online]. Available: https://gerrit-review.googlesource.com/Documentation/intro-user.html.
[41] “Go Contribution Guideline.” [Online]. Available: https://golang.org/doc/contribute.html.
[42] E. Estrada, *A First Course in Network Theory*. OUP Oxford, 2015.
[43] “JSON Flatten GitHub.” [Online]. Available: https://github.com/amirziai/flatten.
[44] “Python Libary Pickle.” [Online]. Available: https://docs.python.org/3/library/pickle.html.
[45] D. Riehle, P. Riemer, C. Kolassa, and M. Schmidt, “Paid vs. Volunteer Work in Open Source,” in *2014 47th Hawaii International Conference on System Sciences*, 2014.
[46] L. MacLeod, M. Greiler, M. Storey, C. Bird, and J. Czerwonka, “Code Reviewing in the Trenches: Challenges and Best Practices,” *IEEE Softw.*, vol. 35, no. 4, pp. 34–42, Jul. 2018.

[47] J. Cohen, “11 proven practices for more effective, efficient peer code review,” *IBM developerWorks*, 07-Nov-2019. [Online]. Available: https://www.ibm.com/developerworks/rational/library/11-proven-practices-for-peer-review/index.html.
[48] G. Bavota and B. Russo, “Four eyes are better than two: On the impact of code reviews on software quality,” in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 81–90.
[49] “Jupyter.” [Online]. Available: https://jupyter.org/.
[50] “Pandas.” [Online]. Available: https://pandas.pydata.org/.
[51] “Numpy.” [Online]. Available: https://www.numpy.org/.
[52] “Matplotlib.” [Online]. Available: https://matplotlib.org/.
[53] “Seaborn.” [Online]. Available: https://seaborn.pydata.org/.