WIRTSCHAFTS
UNIVERSITÄT
WIEN VIENNA
UNIVERSITY OF
ECONOMICS
AND BUSINESS

Master Thesis

Collaboration networks in open-source software development

Jozsef Csepanyi

Date of Birth: 06.09.1996
Student ID: 11927479

Subject Area: Information Systems

Studienkennzahl: h11927479

Supervisor: Johannes Wachs

Date of Submission: 02. April 2021

Department of Information Systems and Operations, Vienna University of Economics and Business, Welthandelsplatz 1, 1020 Vienna, Austria

DEPARTMENT FÜR INFORMATIONS-
VERARBEITUNG UND PROZESS-
MANAGEMENT DEPARTMENT
OF INFORMATION SYSTEMS AND
OPERATIONS

Contents

List of Figures

List of Tables

Abstract

Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi. Aenean vulputate eleifend tellus. Aenean leo ligula, porttitor eu, consequat vitae, eleifend ac, enim. Aliquam lorem ante, dapibus in, viverra quis, feugiat a, tellus....

1 Introduction

In recent years open source software solutions have become widely popular and frequently used in both scientific and enterprise use, which can be attributed to a number of factors, most importantly the ease of development and deployment of IT projects, improved cybersecurity and enhanced scalability [32]. This increases the contribution to open source projects from enterprises and individuals alike. Due to its nature, open source software projects are driven by community contributions, and depend heavily on active participation in all phases of the project.

Software development in a corporate environment usually follows a strict hierarchial structure, where each participant is given a precise position and responsibility, like project manager, scrum master, senior or junior developer, and employees do not tend to work outside of their assigned tasks and territories. The main purpose of maintaining software development structures is for the company to ensure that the outcome of the project is in accordance with the business objective, adheres to the pre-set quality criteria and it is completed in a given timeframe; in other words to asses the risks associated with the business objective of the software project [35]. This is achieved by breaking down the developed software into smaller, less complex components, and grouping the developers into managable teams, where the communication is moderated between teams [6].

As opposed to commercial software development, Free/Libre Open Source Software (FLOSS) projects usually do not follow an organizational hierarchy, and are usually self-organizing and dynamic [6]. Issues, bugs and progress are tracked openly, and everyone is encouraged to contribute based on the current topics and expertise, but purely on a volunteering basis. The lack of access restriction to certain modules allows for much more spontaneous interaction between developers, which generate large, complex networks [24]. These complex networks can be seen as large social networks of developers based on collaboration.

Because contribution to FLOSS projects are voluntary, participants have a different motivation for taking part than in commercial software development. According to El Asir et al. [11], FLOSS participation can be motivated by internal and external factors. Internal factors include self-improvement, learning and contribution as a hobby or pass-time activity [1, 37], whereas external factors are motivated by marketing and demonstrating certain skills, thus increasing and improving employability [1].

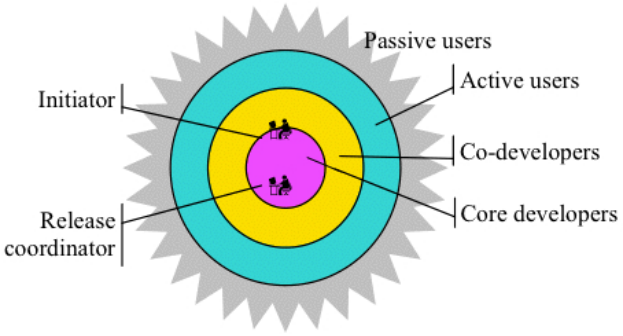


Figure 1: Onion model of collaboration types in FLOSS projects [9].

2 Background and rationale

2.1 Collaboration in FLOSS projects

Collaboration networks of open source software (OSS) have been a subject of many academic research. Raymond [9] has defined collaboration based on bug report interaction, and observed the collaboration network of 124 large-scale SourceForge projects. The generated networks have widely different centralization properties, but it was observed that larger sized projects tend to be more decentralized. The broad community roles contributors tend to take have been also identified in [9], which have been coined as the *onion model* in [24] (Figure 1).

The onion model describes the types of participants in an OSS project as layers. The center represents the small group of core developers, who are responsible for the majority of contributions to the software. They are surrounded by a larger group of co-developers, whose main contributions are usually bug fixes reported by the active users. The passive users are usually the largest in numbers, who do not contribute or report any bugs. In a healthy FLOSS project, each layer of contributors are about one magnitude larger in numbers than the preceeding inner layer [28].

El Asir et al. [11] used a K-means classification to categorise project participants into a similar core-periphery structure (core, gray in-between area, and periphery) based on SNA metrics with a montly timeframe, and analysed how and why contributors transition between groups. They found that technical contributions like code commits and lines added have a much heavier impact on becoming a core developer as opposed to other activities, such as testing, reviewing and commenting.

A literature review conducted by McClean et al. [26] systematically analysed the state-of-the-art research of 46 scientific papers in the field of

FLOSS social networks, and categorised them into three groups based on topic: structure, lifecycle and communication. They conclude, that the existence of core-periphery structure in OSS projects is well established in the field, which is also an indicator of a healthy FLOSS software. Regarding the lifecycle, generally the core development team does not change significantly over time, however, the project becomes more decentralised and distributed as it matures. A lack of research regarding temporal analyses were identified in the most current knowledge, which was suggested as a future research area in this field.

2.2 Social network of Open-source projects

In a larger FLOSS project, developers usually cannot understand every part of the project, therefore collaboration is required with eachother. The network created by the collaborators can be considered as a social network, because collaboration requires some kind of social interaction with eachother. Social network theory describes how social interaction patterns affect the individual behaviour [25]. We can model an OSS project's social network as a graph, where nodes represent collaborators (developers, bug reporters, etc...) and edges represent the social interaction between them. In mathematical terms... TODO

The type of the social interaction determines the created network, therefore choosing the basis of collaboration can have a significant impact on the network structure. The common types of developer social networks (DSNs) are Version Control System-based (VCS-DSN), Bug Tracking System-based (BTS-DSN) networks and DNSs, that are purely based on social elements [2]. The VCS-DSN take the version control application as a source for network generation by recording collaboration based on co-edits of the same module, file or code section. Choosing the granularity can impact the precision of true collaborations represented in the network. Co-edits by multiple developers to a single module or file does not necessarily mean actual collaboration was required from the authors, as the parts edited could work functionally independent from eachother. By increasing the granularity to file sections (classes or functions within a single file) or even lines, we can be more certain, that coordination was required, but we risk leaving out semantically connected parts of the project [18]. In contrast to VCS-DSNs' purely technical approach, the BTS-DSNs use semi-technical bases for connecting participants, such as comments on issues, bugs or reviews [11]. These artifacts, although being tightly related to specific sections of the source code, allow for taking into account conversational elements as contribution. For example, participants, who do not contribute directly to the software source code, but actively review and

comment, are also considered. Lastly, social networks of developers can be constructed on project participation, following, starring or through communication means like mailing lists. The technical aspect of collaboration is minimized in such DNSs, and they are more fit for project organization and communication analyses in FLOSS projects (mailing list vs file file edits vs line edits) ... TODO

2.3 OSS project success

community maturity ([22] in [2])

"Successful projects will likely have modular structure from the start or after refactoring as the source code grows larger and more unwieldy" [3]

success factors: Average Time Efforts, Number of Developers, Comments, Total Code Lines, Comment Ratio, Number of Rater [36] Truck Factor [4]

3 Motivation of research problem and research questions

Because there is a high dependency on the community in open source software projects, by understanding how contributions are included and what patterns emerge we can gain valuable insight into the project's current state and its trajectory. As stated before, SNA analysis of OSS have been extensively studied, but there is a lack of research regarding temporal models analyzing the lifecycle of a FLOSS project.

The goal of this paper is to fill in this gap by examining OSS project collaboration networks over time using SNA metrics. More specifically, one part of the research will focus on the evolution of such collaboration networks and comparing and contrasting these networks with the software outcome. The second part will focus on events during a project, and how it affects the developer collaboration. The research questions, which are broken down into subquestions, are as follows:

1. **How does the temporal lifecycle information of a project influence its success?**

   (a) *Based on temporal models of collaboration, is it possible to predict the outcome of the project?* Since it has been proven that the core collaborators do not change much over the course of the OS software development, our assumption is that any sudden or long-term change, that is not consistent with the other observed

projects, can have a significant impact on the outcome (negative or positive alike).

(b) *Can stages of a FLOSS project with a maturity model be observed?* As most OS software starts with a small collaborator basis and grows over time, it can be assumed, that each project goes through the same steps of open source maturity levels. On the other hand, it is also possible that due to the uniqueness of each project, no such stages are observable.

2. **How do major events in the project lifecycle change the collaboration network of the project?**

   (a) *Do planned or foreseeable events change the collaboration structure?* Major software version releases can be considered foreseeable events of the project lifecycle, which could have an effect on the developer collaboration. For example, there might be a higher rate of interaction between contributors just before a new version is released to clear up the backlog of tasks. But it is also possible, that commit and change rates drop during this time, because the focus shifts to stability and testing instead of new features.

   (b) *How unforeseeable internal or external events affect FLOSS collaboration?* Sudden shocks to the project, such as an announcement of disinterest from major users of the software, discontinued enterprise support of the project, large-scale global events like the pandemic, or sudden employee firings can have significant effect on the core and periphery collaborators alike. By analysing the collaboration network before, during and after such changes, we might be able to recognise patterns, that regularly occur around these events.

3.1 Research methodology

To find answers to the research questions above, first we build a repository analyzer tool, which mines collaboration data from FLOSS projects, generates static snapshot collaboration networks at each given time interval and calculates SNA metrics for each snapshot. Then these metrics can be aggregated over time, or plotted against time to discover changes in the network. The `git2net`[1] [13] Python library provides the necessary tools to mine any

[1] https://github.com/gotec/git2net

project repository that uses git version control. It also incorporates temporal network generation capability, which can be used as a source for creating static collaboration networks aggregated over a given period of time.

We apply a hybrid methodology of qualitative and quantitative research. First, as part of the qualitative research, we choose a small number of repositories to be analyzed. We observe the number of connected components, centrality, number of nodes and mean degree SNA metrics in order to discover the core and peripherial collaborators over the project lifecycles. The basis of collaboration, due to the unavailability of other means of communication, is coediting files. Based on the state of the art research in this field, file coediting proves to be an effective and easy way to represent collaboration between developers.

After discovering the collaboration structure over time, we will match the breakpoints and unexpected spikes or troughs to events within the lifespan of the project. We expect that the key SNA metrics will show a periodicity around planned releases and other reoccurring events (e.g. holiday season). Outstanding values without reoccurrence, on the other hand, are more likely to be consequences of unexpected events. In these cases, it should be observed whether the network is capable of reorganizing itself, or does the event leave a permanent mark on the collaboration structure. A categorization of unexpected events and the level of impact each category has should be observed.

For the quantitative research to be conducted, we will gather a large set of repositories along with major events in its lifecycles. We will then run the miner for all repositories, and with the findings of the qualitative research, we will try to detect all major events and their type (planned or unexpected). We will utilize the `ruptures` [2] library to detect changes in the continuous SNA metrics. If the model is capable to accurately recognise events, then we can also apply it on any repository to detect changes, which will allow us to discover changes in the collaboration network that are not related to publicly known events or releases.

4 Gitminer implementation

To find answers to the research questions, we implement an analysis tool to mine and analyze project repositories, which allows us to generate collaboration networks and network metrics for the analysed projects.

[2] https://github.com/deepcharles/ruptures

4.1 `git2net` miner

The process begins with the project mining. After cloning the repository, the `git2net` [13] library is used to collect data related to commits. Specifically, who is the author of each commit, which files were modified (created, edited, deleted) with the commit, and when was the commit created. Additionally, the lines edited by the author within each commit are collected separately, allowing for a more fine-grained collaboration network generation if necessary. The results are collected into an `sqlite` [3] database file's *commits* and *edits* tables.

The `git2net` mining process by default collects all the commits throughout the project's lifecycle. However, the processing time of each commit differs based on the number of edits, the affected number of files and the file types as well, which makes collecting certain commits very resource-intense and time-costly. Therefore, we exclude every commit, which contains more than 100 file modifications, during each repository mining using the *max_modifications* parameter. As observed by Gote et al [13], this exclusion criteria does not affect significanly the generated network, because they are mostly merge commits or project restructurings, which do not mark any true collaboration effort between developers. During the data mining in certain repositories, we encountered commits, that were not mineable with this method and the mining process halted, presumably due to processing error because of binary file changes in these commits. We also excluded these commits from our data mining process.

This exclusion criteria resulted in an average of 3% of commits excluded in all repositories subject to our analyses, with the highest excluded commit rate being 20%.

4.2 `repo_tools` miner

We use the `repo_tools` [4] Python library to query the Github API for additional repository data extraction, such as:

- Releases
- Tags
- Issues
- Stars and followers

[3] https://www.sqlite.org/index.html
[4] https://github.com/wschuell/repo_tools/

The mining output is also stored in a `sqlite` relational database, which is queried later on during the analysis.

4.3 Data preprocessing

The collaboration networks with the `git2net` library connect the authors to their edited files using only file and author names instead of IDs. This creates an issue when generating the networks, because authors with the same name will show up as one node, and they will be connected to the files they touched combined. Furthermore, authors that change their displayed name ('author_name' field in the mining database) or log in from diferent accounts, where they have different names, will show up as multiple nodes instead of a single vertex.

We utilize the `gambit` [14] rule-based disambiguation tool to resolve the author names. Furthermore, the created networks have issues when the node names contain special characters or spaces. Therefore, after disambiguation, we replace every unique author name with its ID number.

As the files are also labelled by their filename property in the network outputs, the same filenames but in different folders are also displayed as single nodes. In order not to create false collaborations, we simply remove the files from the network with filenames, that occur more than once in all the repository subdirectories. We argue that this does not remove any significant collaboration data, since most files sharing their name with other files are technicaly files, like `__init__.py` for a Python project.

4.4 Collaboration networks

When creating a DSN from the mined data, we have multiple methods at hand. The `git2net` library provides its own co-editing network function, which returns a temporal network of collaborators. This uses the co-authorship algorithm developed by Gote et. al. [13], however, we would like to have more control over the network generation method, such as simple file-based co-authorship in order to customize the network for our needs, like weighing each relation or generating undirected graphs.

4.4.1 Temporal bipartite network

As a first step, we generate a temporal bipartite network of authors and their edited files with the `git2net` built-in *get_bipartite_network* method. A temporal network is a `pathpy` [5] graph object, which contains a collection

[5] https://www.pathpy.net/

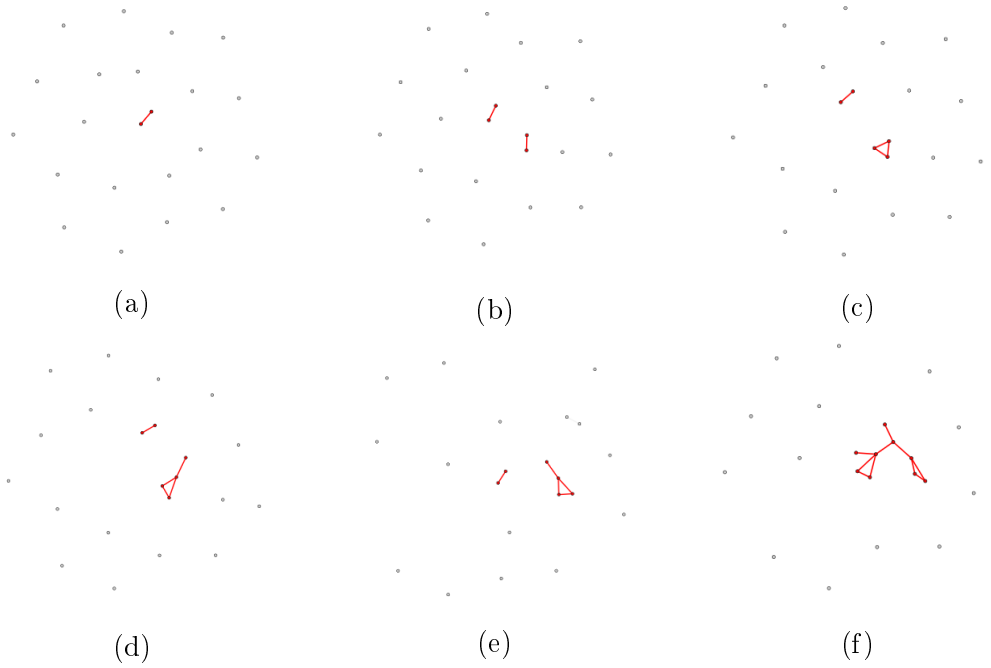


Figure 2: Sequential snapshots of the `networkx` collaboration network with a moving time-window of 30 days and 7-day steps.

of timestamped graphs of a single network at each point in time within the observed timeframe. Such a snapshot $S_t = (U, V, E_t)$, where U is the set of authors, V is the set of files and E_t is the set of file edits as edges at t timestamp. By connecting the authors, who touched the same files, and removing the nodes representing the edited files (converting the bipartite network to a regular network), we can observe the evolution of the collaboration over time, represented in Figure 2.

Although a temporal network preserves the time aspect of the graph by the edges being tied to the time dimension of the graph, calculating network metrics like centrality on such networks is infeasible. Visualization also proves to be difficult in representations where animation is not possible. Therefore, we aggregate the bipartite network over a given timeframe into a static network. All nodes within the temporal net are preserved, and all directed edges are added to the network with the edge weight representing how many times that author edited the file.

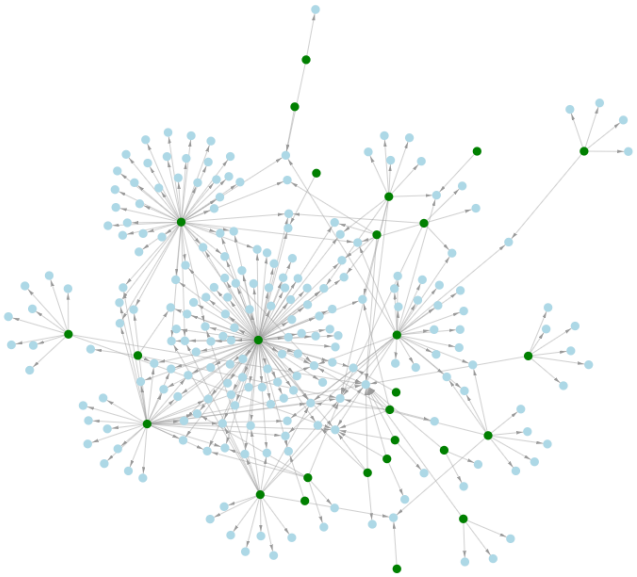


Figure 3: A bipartite network of authors (green) and edited files (light blue) in the `pandas` project within the timeframe 31/01/2021 and 15/02/2021

4.4.2 Static networks

The generated static weighed bipartite network looses its time-varying component, but now we are able to manipulate and calculate complex statistics over it. Figure 3 is an example of such a network. As a next step, we convert the the bipartite network into an authors' network by removing the nodes representing files.

We have multiple methods to convert the directed and weighted bipartite network into a projection of authors. We could simply remove the files and connect each author, that worked on the same file, however, the end result would be an unweighted graph. This would falsely show, that all collaborations are weighed equally, which is clearly not the case, as multiple continuous edits on the same file from both parties should represent a stronger collaborative connection. Therefore, firstly we implement the Weighted One-Mode Projection (WOMP) method [34], The WOMP method converts the bipartite network $G(A, F, E)$, where E is the edge list containing tuples (a_i, f_i, w_{ij}), and $w_{ij} \in E$ is the weight between author $a_i \in A$ and file $f_i \in F$. With this notation, a weighted directed edge can be calculated for any $a_a, a_b \in A$ as follows:

$$w_{ab}^{A \to A} = \sum_{j=1}^{m} \frac{w_{aj}}{W_a^F},$$

where W_a^F is the sum of all outgoing edge weights from author a to all files F denoted as $W_a^F = \sum_{i=1}^{n} w_{ai}$. This creates a bidirectional weighted collaboration network between authors a_1 and a_2, where the weight w_{12} represents the relative collaboration effort of a_1 towards a_2 compared to all the other developers a_1 has collaborated with. Conseqently, every edge is in the rage $[0, 1]$ in the resulting WOMP network.

A disadvantage of the WOMP method is, that the generated collaboration network is bidirectional, meaning if there was any common authored files between a_1 and a_2, then there will be both w_{12} and w_{21} connecting them. To simplify the network, we want to generate an authors network, where the edges are undirected. For this, we are using the weighted Jaccard method on the files-authors bipartite network:

$$w_{ab} = \frac{\sum_{f \in F} min(f_a, f_b)}{\sum_{f \in F} max(f_a, f_b)}.$$

For each file f that a_1 and a_2 authors touch, we sum up the minimum and maximum weights the authors have towards each file, then we divide the sum of minimums with the sum of maximums. This results in the undirected author-author network with edge weights in range $[0, 1]$. By default, this method removes isolated contributors, who do not collaborate with eachother, but are actively editing the files. We add these nodes manually. Figure 4 shows the final author network.

4.5 Core and periphery, centralization

A critical part of the OSS software projects is the existence of core and periphery developers. It has been observed, that in each FLOSS project there are a small number of developers, who provide the vast majority of development effort into the project. It has been also established, that the members of core developers do not change substantially during the project's lifecycle. However, there was no effort on whether there is a change in the collaboration pattern, especially before, during or after a major lifecycle event. Therefore, we make efforts identifying the core developer network to observe these changes.

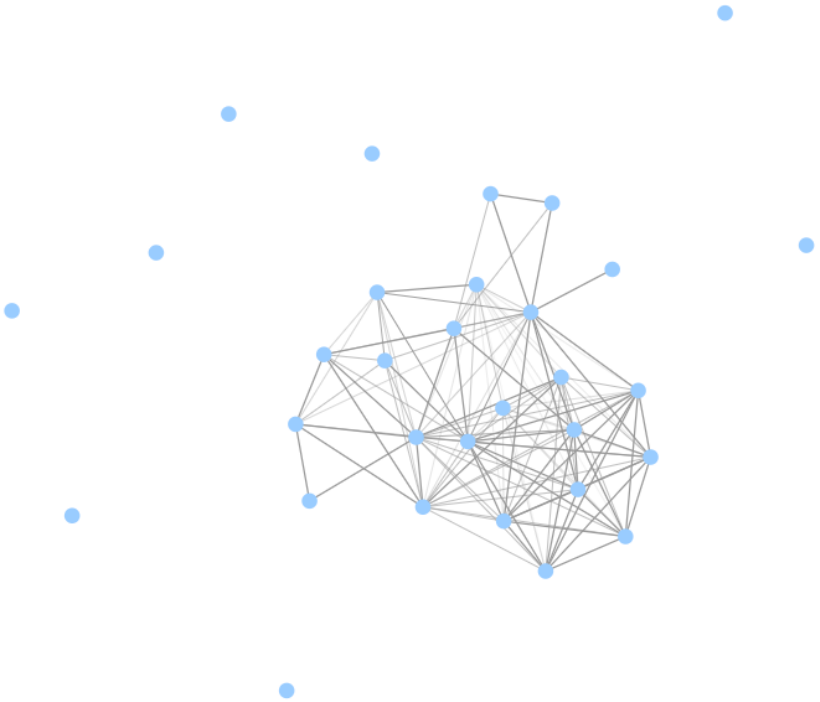


Figure 4: Weighted Jaccard similarity collaboration network of `pandas` generated from the bipartite network in Figure 3.

4.5.1 Degree centrality

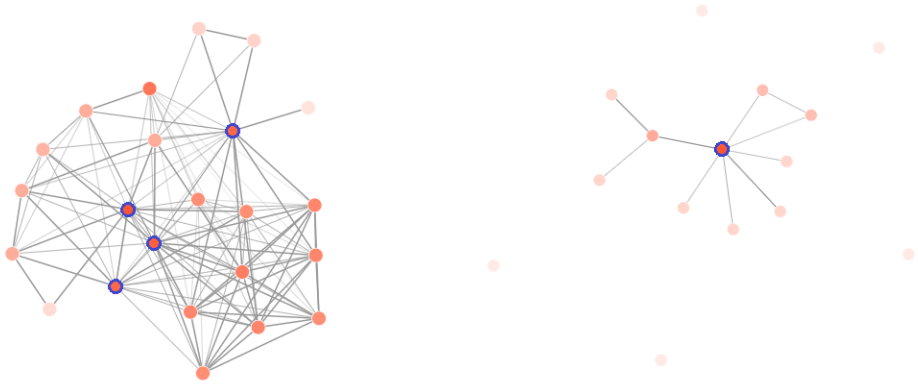
We use the degree centrality of each node (i.e. developer) to identify the core members. Degree centrality of a node is the fraction of all possible nodes it is connected to. We can calculate it by dividing the degree with $n-1$, where $n = |G|$ the number of nodes within the network. Since the core developers contribute the majority of commits and edits of the project, they are expected to be connected with more nodes. Joblin et. al. [17, 18] have also identified degree centrality as the best predictor of core developers. In cases, where binary classification of core or periphery is needed, we assign developers to the core network if their degree centrality score is in the top 20th percentile, otherwise they are considered as periphery. We also take note, that this method does not consider the weighted edges, only the number of edges (degree) a node has. Although this method could be refined to consider the node degree weighted with the edges, we argue that this could lead to invalidity. In case of two developers, who only contributed to one file, they will be represented with a strong connection and would receive a high weighted degree value, whereas a core contributor, who edits many files, can have many weak connections but these might not add up to one strong connection of the two isolated developers when weighted with the edge weights. It is clear, that a developer with many connections, regardless of the strength of the collaboration, should be considered core. Figure 5 shows two examples for degree centrality within a collaboration network. In the figure, darker colors represent a higher degree centrality value. The highlighted nodes are in the highest 20th percentile of degree centrality, classifying as members of the core developers. We can observe in both one-month periods, that `pandas` is much more decentralized, whereas `curl` is largely dependent on one developer.

4.5.2 Degree centralization

The degree *centrality* can be calculated for every node, however, through our analysis we would also like to measue a global *centralization* metric, which is applicable to the whole network. As suggested by Crowston and Howison [10], we calculate the degree *centralization* by summing the differences between the maximum and each node's degree *centrality*.

$$C_D(A) = \frac{\sum_{i=1}^{n}(C_d(a*) - C_d(a_i))}{H},$$

where $C_d(a)$ is the degree *centrality* of an author a, $a*$ is the author with the highest degree *centrality* value, and n is the number of authors in the



(a) Pandas

(b) Curl

Figure 5: Degree centrality within the `pandas` and `curl` projects' collaboration networks.

collaboration network A. The value H is for normalizing the sum by dividing by the theoretical maximum *centralization*. Since the *centrality* values are already in the range $[0, 1]$, we only need to normalize for the network's size. We get the highest centrality score with a star graph, where each node is only connected to a single central node, which has exactly one edge to all other nodes. The central node has a centrality of 1 in this case, whereas all the other $n - 1$ nodes have $C_d(a) = \frac{1}{n-1}$. This means that in case of a star graph:

$$H = (n-1)(1 - \frac{1}{n-1}) = n - 2.$$

Within certain timeframes, when a project is inactive, it could happen, that the network contains 2 nodes or less. We define $C_D(A) = 0$ if $|A| = n <= 2$. The resulting output will always have a value in $[0, 1]$, where 1 means a completely centralized network (star graph) and 0 means a completely decentralized network. It is important to emphasize that a centralization score of 0 does not necessarily mean that there is no collaboration and every developer is isolated. Rather it means that each developer is co-authoring with just as many authors, as the others do.

4.5.3 Clustering coefficient

While centralization helps us describe the centralness of the network and how much it is centered around a single, or a small number of developers,

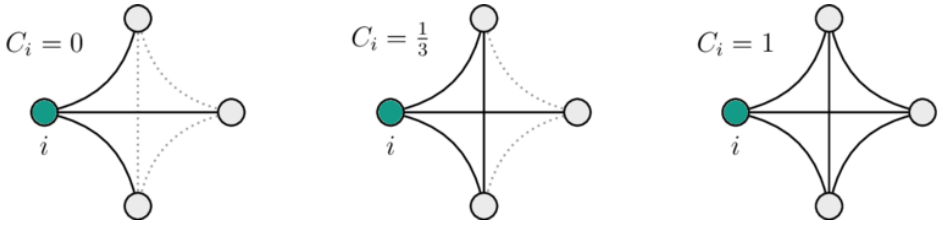


Figure 6: The local clustering coefficient demonstrated on an unweighted network of 4 vertices [16].

it does not help us describing the structure of the network in more detail. Our goal is to gain an understanding of also the modularity of our network, meaning how much developers tend to cluster together [18]. We expect that authors form smaller clusters, which are more tightly connected together, and these clusters have somewhat weaker ties to other clusters. This builds on the assumption that the social network of the software follows the modules which build up the software itself, thus authors of a specific function should also cluster together within the network [8, 18]. To measure this "clusteredness", we calculate the *local clustering coefficient* for each node.

The *local clustering coefficient* quantifies on a scale $[0, 1]$ how likely it is that a node's neighbours are also neighbours. We use the number of how many triangles (also called clique, triplet) is every node a part of. This is illustrated for unweighted networks on Figure 6 with the formula:

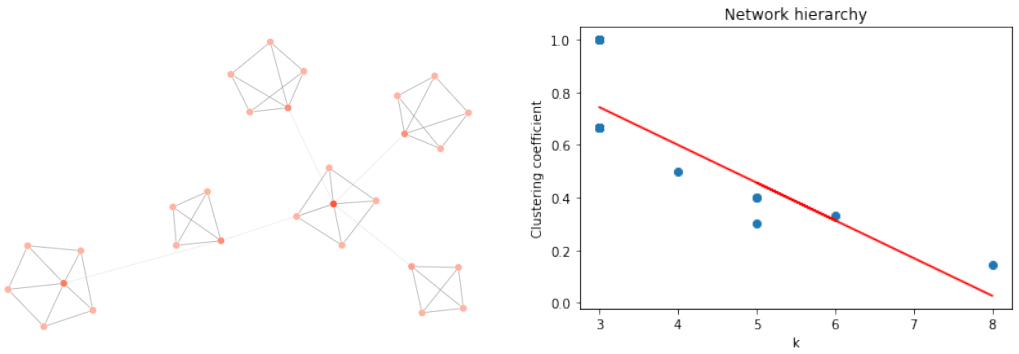
$$C_i = \frac{2T(i)}{deg(i)(deg(i) - 1)},$$

where $T(i)$ is the number of triangles through node i and $deg(i)$ is the degree of i. However, in a weighted network we also have to consider the edge weights, since it is easy to see that a clustering coefficient of 1 with also the maximum weighted edges in a triplet does not represent the same clustering as being connected with a weak links. We expect weaker links connecting larger clusters, whereas stronger links within each cluster. Therefore, we use geometric averaging of the subgraph edge weights (as implemented by the `networkx`[6] library [30]):

$$c_i = \frac{\sum_{jk}(\hat{w}_{ij}\hat{w}_{ik}\hat{w}_{jk})^{1/3}}{deg(i)(deg(i) - 1)}.$$

The $\hat{w}_{ij}$ represents the normalized weight of edge e_{ij} over the maximum weight in the network.

[6] https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.cluster.clustering.html



(a) Highly hierarchical network

(b) Network hierarchy

Figure 7: Hierarchical network and its corresponding degree number vs clustering coefficient plot.

4.5.4 Hierarchy

The degree centrality and the clustering coefficient are in themselves able to express meaningful aspects of the developer social network, however, by combining the two metrics, we can also assess how hierarchical the network is. In a scale-free social hierarchical network (such as the collaboration network), nodes tend to cluster around a single or a few hubs, which are more likely to have weak connections to other hubs [33, 18]. The nodes within these formed groups are relatively stronger than the connections connecting the hubs, but they are less likely to connect to nodes outside of their group. Therefore in a hierarchical network, the hubs have a high degree number and a low clustering coefficient, whereas the group members clustering around the hubs have a high clustering coefficient, but low degree numbers.

We can visualize the degree of hierarchy by plotting each node's clustering coefficient against the number of degrees, shown in Figure 7. In hierarchical networks, the plotted linear regression trendline will decrease steeply, as there is a negative correlation between the degree and clustering coefficient. In networks, where this cannot be observed, the trendline stays flat, meaning these two metrics are independent from eachother and the structure is not hierarchical. To measure the hierarchical level numerically within a network, we take the trendline's slope, which is β_1 in the $y = \beta_1 x + \beta_0$ general linear regression equation.

In every network isolated authors can be observed, who are only working on files that noone else has edited (in the given timeframe). These nodes can skew the hierarchy score, because an isolated node's degree and clustering coefficient are by definition 0, which disproportionately makes the trendline

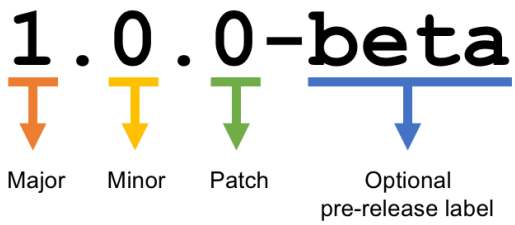


Figure 8: Semantic versioning. Figure source: [23].

much flatter. Therefore, we remove the isolated nodes from the network. If the network only contains isolated nodes, and there is no linear regression to be calculated, we set the hierarchy value to 0.

4.6 Project measures

One of our assuptions is that the collaboration network structure changes depending on the project's lifecycle. In order to discover cause and effect relationships between the network structure and project lifecycle, we gather basic project metrics to pinpoint the time and date of events, as well as the effort required within the project. We achieve this by gathering the dates of each release, and the quality and relative stress is measured by the issues within the project (create and close times).

4.6.1 Release and release measures

To measure the network changes around releases, first we collect the list of releases for the given project. Our goal is to gather the version number of each release, as well as the dates they were released. There are two relevant APIs regarding the release version numbers: GitHub releases and Git tags. Tags are marked and annotated commits supported by Git, therefore other projects, that are not on GitHub can also have tags. *Tags* are most commonly used to mark new release versions, but they can also be used to annotate other information, such as release editions or milestones. On the other hand, a *Release* is a high-level GitHub concept, which allows the project organizers to announce Git tags as project releases by adding a version number, release notes and binary artifacts [29]. A commit, that represents a new GitHub release, must be tagged, but a tagged commit does not necessarily need to be a new GitHub release.

Throughout our analysis, we use the tags as indicators of a new release, because there could be new versions of a software, that are not released publicly, and therefore do not have a GitHub release. Furthermore, GitHub

only announced the Releases workflow in 2013, when the release versioning by Git tags was already a common practice. This means, that releases before 2013 can only be analysed via the repository tags.

Most large-scale OSS proejcts follow the semantic versioning convention, but only to a certain extent. The majority of tags follow the major-minor-patch (also known as breaking-feature-fix) semantic version naming convention in the format of X.Y.Z, where X is the major number, Y is the minor number, and Z is the patch number [31]. At the end, optionally pre-release and build data can be marked, for example: *1.11.6-pre*. The major number signifies an API-breaking change compared to the previous release, meaning backwards compatibility is not guaranteed for depending applications. Minor releases add new features, but compatibility is ensured with the older version. Patch releases usually handle bugs and security updates within the package.

During the network analysis, we expect the network measures to change around a release, but it is also expected, that a larger release, that required more collaboration, will have a greater impact, while smaller changes have smaller or no impact at all. The issue with the semantic versioning and release names is that there are no constraints, which would enforce a strict version naming, and it is entirely up to the developers to set the tag names. This leads to inconveniences when measuring collaboration effort through release version number, because a patch might require more collaboration effort than a minor release, and a minor release within one project could require significantly more teamwork than in another. Furthermore, tag names can be inconsistent, and there could be version numbers, that do not adhere to the major-minor-patch naming convention at all (e.g. test releases, or releases like 'latest-release-v11'). The possibility to add extra information at the end of tag names, like `-beta` or `rc` further complicates tracking the collaboration effort, because the majority of collaboration effort might happen before the `rc` version, or it might happen after it. As it can also be seen in Table 1, the `v0.14.0rc1` tag contains more modifications than the final `v0.14.0` version, whereas the `v1.2.0rc0` only contains a small fraction of modifications compared to `v1.2.0`.

In order to have a more fine-grained measure of how much effort a release required (besides the semantic versioning), we measure the number of lines added and lines removed in that release. This is calculated by adding up each commit's 'total lines added' and 'total lines removed', which was authored after the previous release but up to and including the current release tag's commit. The total change of lines for a release is simply the sum of lines added and lines removed, which are provided by the `git2net` miner. The miner also provides the number of modifications for each commit. A modifi-

Index	Name	Tag name	Created	Type	Modifications	Lines added	Lines removed	Total change
0	Pandas v0.13.0	v0.13.0	2013-12-30 17:02:51	unknown	20031	1.665357e+09	1.624343e+09	3.289700e+09
1	Pandas v0.13.1	v0.13.1	2014-02-03 04:52:01	patch	836	2.231875e+06	2.349833e+06	4.581708e+06
2	Pandas v0.14rc1	v0.14.0rc1	2014-05-16 22:28:09	minor	1861	2.251189e+07	2.050846e+07	4.302035e+07
3	v0.14.0 final	v0.14.0	2014-05-30 11:47:40	unknown	318	3.220804e+06	5.314470e+05	3.752251e+06
4	v0.14.1 final	v0.14.1	2014-07-10 23:46:19	patch	720	1.660133e+06	3.796690e+05	2.039802e+06
5	v0.15.0 Pre-release	v0.15pre	2014-09-07 12:52:01	unknown	826	1.028779e+07	2.458664e+06	1.274645e+07
...	...	...	...	...	...	...	...	...
66	Pandas 1.1.5	v1.1.5	2020-12-07 11:42:10	patch	2167	4.251057e+06	4.611365e+06	8.862422e+06
67	Pandas 1.2.0rc0	v1.2.0rc0	2020-12-08 12:31:44	minor	41	1.060500e+04	8.850000e+02	1.149000e+04
68	Pandas 1.2.0	v1.2.0	2020-12-26 13:47:00	unknown	683	5.782570e+05	2.989310e+05	8.771880e+05
69	Pandas 1.2.1	v1.2.1	2021-01-20 11:21:02	patch	1306	1.864636e+07	9.228629e+07	1.109326e+08
70	Pandas 1.2.2	v1.2.2	2021-02-09 10:55:19	patch	844	6.783283e+06	1.827142e+07	2.505470e+07
71	Pandas 1.2.3	v1.2.3	2021-03-02 09:43:36	patch	959	1.002189e+07	5.783745e+06	1.580563e+07
72	Pandas 1.2.4	v1.2.4	2021-04-12 15:59:13	patch	289	3.402820e+05	1.704960e+05	5.107780e+05

Table 1: Releases collected information.

cation is a section of the source code modified, which can mean multiple lines added and deleted at the same part of the document. For example, if a new function is added to the project, which requires 20 new lines and removes 2 lines (e.g. empty space that was there before), then it will be considered as 1 modification, but 22 total line change. Changes to binary files are due to generated artifacts, which do not carry any collaboration effort, therefore they are excluded, and commits, that do not have a hash are also removed.

The release type contains the semantic version of the release, which was gathered from the tag name with a Regular Expression matching the conventional versioning X.Y.Z. We also capture the version number in tags, that contain additional notations such as `beta` or `rc`, then we compare the current release to the previous to identify whether the release is a major, minor or patch release. When the found version numbers in the previous and in the current release are the same, we leave that as unknown, as this is mostly the case in pre-releases. The first release doesn't have a preceeding tag, therefore we consider every modification before the tag as part of the release. This leads to the first tag seeming to have significantly more edits than the rest of the releases, whereas in fact this is just the result of not tracking from the beginning (see example in Table 1). Therefore, in our analyses we remove the first release, as this would lead to falsely weighing the network results.

4.6.2 Project issues and measures

To measure the productivity within the project, we collect some basic information regarding the GitHub issues. Issues keep track of bugs, beatures and tasks, contributors can comment and discuss the task at hand within an issue, and it can be assigned to users and milestones can be set. We collect the following information of issues:

- Issue title

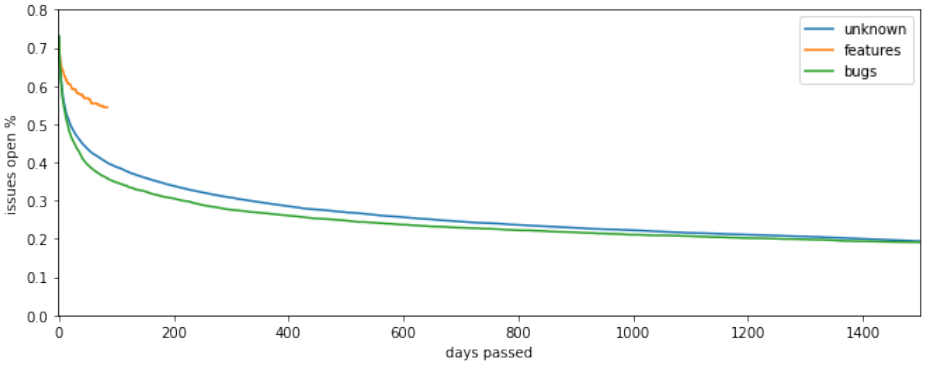


Figure 9: Survival curves of the `pandas` library.

- Issue number
- Created at
- Closed at
- Open for
- Bug or feature

The *issue title* is a short description of the issue. Most projects create their own convention of naming and tagging issues, for example each issue starts with a 3-letter abbreviation of a category, e.g. `BUG` or `DOC`. *Issue number* is a unique number for each issue, that is increased sequentially. *Created at* is the date and time of the issue being created, and *closed at* is the time when it was closed. If the issue was still open on the day of data mining (May 9 2021), this field is empty. The *open for* field is the difference between the *closed at* and *created at* dates, or it is empty if the issue was never closed.

The last measure is *bug or feature*. Because issues can cover a wide range of possible topics, from discussions to performance, it is worth categorizing them to analyse the differences. However, categorization in practice proves to be difficult due to the different conventions each project uses, and because GitHub does not apply any constraints to the text of the title. We are searching for specific keywords in the *issue title* to categorize each issue into either a bug or a feature. We also drop some of the words from the text, because they would give us 'false positive' matches. If keywords for both or neither categories are found, the issue is marked as unknown. The dropped words, as well as the bug and feature keywords can be seen in Table 2 Although this simple method can already classify 10-15 percent of issues,

Drop words	Bug keywords	Feature keywords
debug	bug	feature
debugger	defect	enhancement
debugging	incorrect	improvement
	unexpected	suggestion
	error	wishlist
	missing	wish list
	warning	
	problem	

Table 2: Keywords and drop words for *bug* and *feature* categorization.

it would require extensive manual effort to further improve this ratio. In future research, sentiment analysis of issue titles and issue descriptions could achieve much higher percentages.

Statistics for issues like average close time also prove to be difficult, because there are continuously open issues, which inevitably leads to the fact that a large portion of issues were still open and continue to be open. For these issues, we cannot know the issue close time, and if we try to calculate global measures by aggregating all (or portion) of issues, we have to consider the *survival bias* [15]. Figure 9 shows the issues survival curve of the `pandas` library. It is clearly visible, that a large portion (20 percent) of issues are not closed, and if we left them out of aggregate values such as average open time, our data will be skewed. Nevertheless, we can see in the example, that the categorization of issues makes a difference, as bugs get closed a bit earlier than an average (unknown) issue, because after the same amount of days passign for both categories, more bugs tend to be closed in this example. In contrast, features have a high life expectancy, as they are more likely to be still open than a bug, if the same number of days pass. The reason could be that features take longer to develop and plan than bugs, which is consistent with the findings of Jarczky et. al. [15].

5 Collaboration pattern analysis

The focus of our analysis is how the collaboration network of FLOSS projects changes during their lifecycle. Moreover, we are also interested in the cause of the changes, rather than just observing the changes. Therefore we determined two types of events, that we want to observe in order to see the effects on the collaboration. The events during a project's lifecycle can be one of two types: regular or irregular event. Regular events repeat over time

with roughly the same time passing between events. This can be a holiday season (e.g. summer holidays or at the end of the year) or regular software releases. In our research, we will focus on only the software releases, and we suggest the analysis of collaboration and seasonality as a future research topic. In contrast to releases, which are usually well documented and easily discoverable when exactly they were released, irregular events are harder to discover, as these do not necessarily stem from within the developer community, but from external sources. Examples could be a supporting company's organizational restructuring, the sudden increase of work from home during the COVID pandemic, or mass layoffs within the support team. In our analysis, we will mainly focus on layoff events, because of our expectation that this will have the largest effect on collaboration. Usually when a company develops an open-source software, the company relies on the external community of the project, but the core developers are mainly from the company. Therefore, when a company officially pulls out from further development, this results in the removal of most core developers in a short amount of time. Our main goal is to identify how this affects the collaboration network, how the network restructures itself, or if any restructuring occurs at all.

As a first step, we conduct a qualitative analysis on a number of handpicked repositories to observe the network statistics in detail and to identify cause and effect relationships. In order to avoid bias when selecting repositories, we choose projects based on a variety of factors:

- *Project size:* We define the project's size as small, if it has less than 100 contributors, medium sized if the number of collaborators are between 100 and 500, and large if it has more than 500 collaborators.
- *Centralization:* Although it is hard ot measure how decentralized a project is, based on the literature review, very large projects tend to be decentralized. For now, we consider those projects centralized, which have been identified as centralized in the current state-of-the-art literature.
- *Event occurrences:* Whether there is a regular release cycle within the project or not based on the release dates, or if there are known and confirmed cases of unexpected layoff events.

Based on these criteria, we try to select a wide variety of projects, so that each type (small, medium or large size, centralized or decentralized, regular or irregular or abandoned projects) has at least one example. We acknowledge the fact that this inclusion criteria leaves room for selection bias, as an observed correlation between events and the network in one project

Name	Contributors	Size	Commits	Stars	Issues	First release
pandas	2333	large	26792	29700	3518	Feb 20, 2011
numpy	1135	large	26392	17200	2030	Jan 5, 2002
networkx	469	medium	6470	9100	166	Jul 17, 2005
seaborn	140	medium	2780	8400	82	Oct 28, 2013
curl	701	medium	27159	20600	27	Mar 14, 2000
servo	1101	large	44084	19600	3305	May 22, 2017
wasmtime	243	medium	8334	5200	341	Oct 18, 2016
py-junos-eznc	69	small	2486	583	76	Nov 3, 2013

Table 3: Collaboration analysis projects and basic statistics.

might not hold true for all projects. However, later we confirm our hypotheses with a quantitative analysis.

5.1 Project selection and descriptions

All together we choose 8 projects to analyse, these can be seen in Table 3. Due to the repository miner's limitation, we are only considering projects maintained on GitHub. The `pandas` and `numpy` repositories are well-known and popular data science tools for data manipulation in Python. They are very similar in size and popularity, both with a huge base of collaborators. Since also the number of issues are really high, our expectation is that these projects are highly decentralized, where the developers mainly work on their own parts of the project. They also have fairly regular release cycles, with a minor release in every 5-6 months.

`Networkx` and `seaborn` are both middle-sized repositories based on the number of contributors, and they are both used for data visualization and statistical analysis in Python-based software, but `networkx` visualizes networks and their related statistics, while `seaborn` creates various plot images. Each release date follows the previous with 4 to 10 months, which means their releases are not as regular as the first two repositories.

The project `curl` is an example for a highly centralized project, as its source code is famously maintained by a single developer [10]. Based on the number of contributors, it is a medium-sized project, although the number of commits are much higher than `networkx` or `seaborn`. Being the oldest project from the selection might contribute to the large number of commits.

`Servo` and `wasmtime` are both based on the Rust programming language, and they are both used for web applications: `Servo` is a browser engine, whereas `wasmtime` is a runtime environment for WebAssembly. Both projects were led by Mozilla and maintained in an open-source environment. The Rust developer team was heavily affected by the Mozilla layoffs on January 15$^{\text{th}}$, 2020 and August 11$^{\text{th}}$, 2020 [21, 19], with the second round of layoffs af-

fecting all Rust employees. The `py-junos-eznc` project is considered small, since it has less than 100 contributors, and it is a Python-based library for automatizing devbices running on Juno OS. On May 27th, 2020, the sponsoring company Juniper Networks laid off its entire open source developers [7]. These three projects will be the main focus of analysing unexpected, one-time events and layoffs, where we will take a close look at the network statistics around the mentioned dates.

To reliably compare the projects, we have to consider only a slice of each project, so all of them are compared within the same time period. We take a 3-year period from 2018 to the end of 2020. All projects had their first release before this period, which means this should filter out the initial irregular activities, such as creating directories and restructuring.

5.2 Commits analysis

For the selected period, we check the number of commits in each project as a first step. This can be seen on Figure 10. To smooth out the stacked area plot, we summed up the number of commits in every 28 days. Purposefully a multiple of 7 was chosen in order to even out the possible irregularities caused by the weekdays and weekends. We can see that all projects are active, with varying magnitude of commits generated each month. The number of commits are consistent with the size of the project, but we can also see that the number of contributors and the number of commits are not completely dependend on eachother, as `servo` consistently receives more commits than the other large repositories, despite the fact that it has the same number of contributors as `numpy` and half as many as `pandas`. Our assumption is, that this is highly dependent on the type of software being developed, the used programming language(s) and on other project-specific properties.

There is a noticeable decrease in the number of commits to `servo` at the time of the layoffs (August 2020), which signals that the project could have been affected. This cannot be observed for `wasmtime`. We do not see any evidence of reoccurring trends within the number of commits, e.g. decreased activity during holiday season. There is a reduction in all repositories at the end of 2018, but this is contradicted by a spike in the number of commits at the end of 2019. However, we cannot rule out any periodicity, because this could be hidden due to the 28-day aggregation.

5.3 Releases

As a next step, we take a look at the regular lifecycle events, which are the releases in the open source projects. We generate the collaboration networks

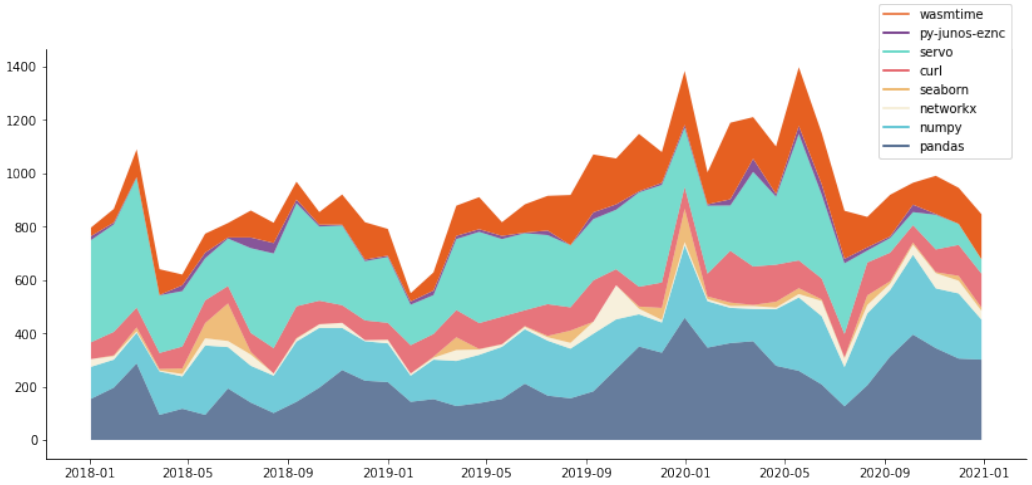


Figure 10: Number of commits in each month.

for the same time period (between 2018 and 2020) and contrast them with the time of releases.

The issue with generating the collaboration network is the time aspect: we cannot observe any collaboration network if we take a single point in time and create the network for a given timestamp, as this will only contain a handful of connections, who authored an edit in the same file at the exact same time. Therefore, we need to aggregate the edits and commits over time, which requires us to use a time window to scan through the observed time period, and for each step, we generate a snapshot of the network. The first parameter is the time window length and the second is the step size. By choosing a large time window, we observe more connetion within the generated graph, but with a too large value, the network becomes too much grouped together, with almost every node connecting to almost all other nodes. By setting this value too low, the result is a disconnected network. The step size determines how often we take a screenshot of the network for the time window. By default, we use one-day steps, but sometimes this results in too radical changes. Therefore in some cases we use a 7-day or a 28-day step value to smooth out the curves. However, a too high step value can obscure the changes we want to observe.

5.3.1 Node counts

First we take a look at the number of developers in the network over time. The number of nodes (i.e. developers) in the network indicates the level of activity within the project, where a relatively high number of nodes imply a

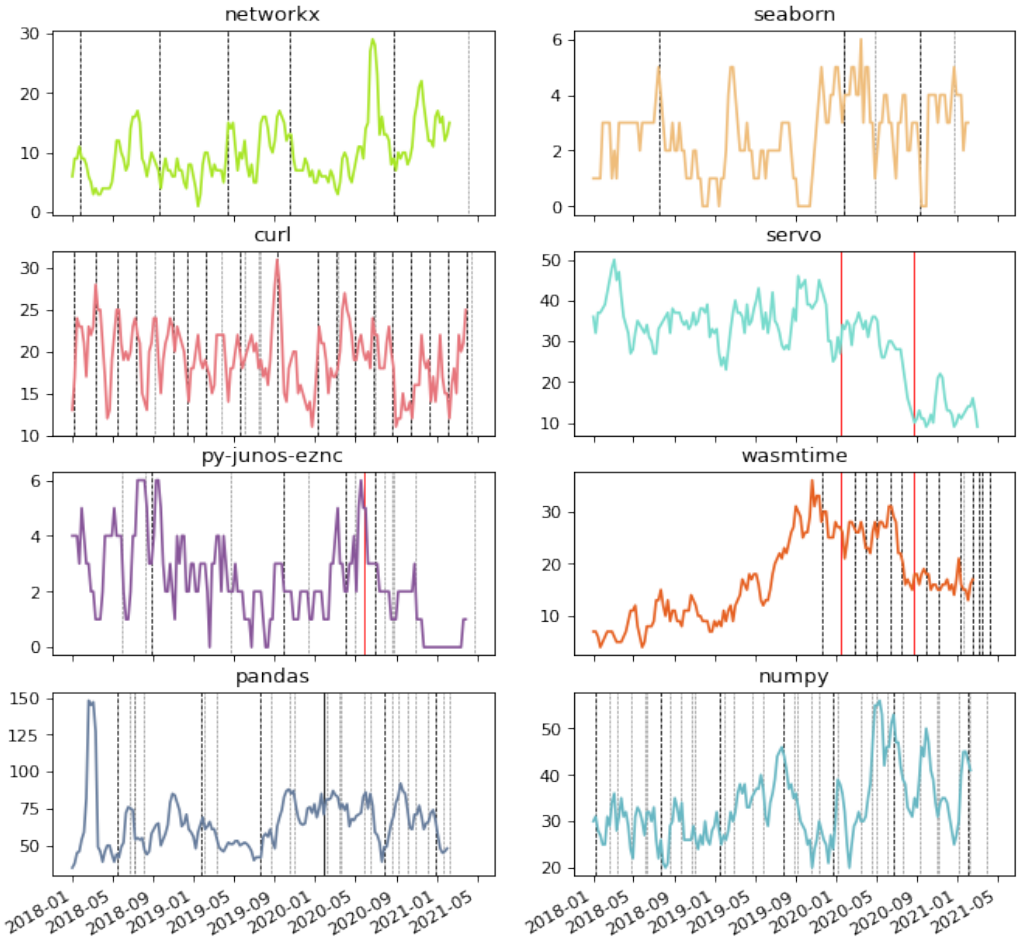


Figure 11: Number of nodes over time.

peak in project activity, and a low number can imply a decreasing activity, or that the project has become more centralised. It is easy to see, that if the only source of activity is one single developer over the observed period, then the network will only have one node, whereas if the same activities are distributed, more developers will be shown in the network, and the number of nodes will rise. However, this does not describe the relationship between developers, they could be completely isolated, or they could be relying a lot on eachother regarding collaboration.

In Figure 11 the number of nodes are plotted with the time of releases marked with a vertical black line for each project. The type of release corresponds to the type of line: major releases are shown with a continuous line, minor releases are thick dashed lines, and patch releases are thin dotted lines. The red continuous lines mark an unexpected layoff event, which might had

an effect on the project. Each node count data point at t time represents the number of nodes within the collaboration network that is generated between t and $t + 28$ days, with a 7-day step size. The average node counts correlate with the size of the project, as the smaller repositories like `py-junos-eznc` and `seaborn` have the number of nodes consistently below 10, while the larger projects can reach 30, 50 or even 100 at certain time periods.

Regarding the releases, we can see that each project can have completely different release planning and schedule, which makes it hard to compare them. Some projects, like `networkx`, `pandas` or `numpy` release a minor version in approximately every 6 months, while `curl` does the same in every 2 months. `wasmtime`'s release schedule regaring minor releases is even shorter, but we do not have any data available before 2020, which does not allow us to confirm whether this is a long-term trend. The release data for `servo` was available to us, however, the repository has a very high frequency of releases, with sometimes multiple minor releases in the same week, and displaying this information on the chart would have obscured the number of nodes plot, therefore we only display the unexpected events. Also, releases can be arbitrarily named by the developers, therefore a minor release in one project might be considered a patch or a major release in others.

Comparing the events with the number of nodes, we can observe, that before or around a minor release, most projects experience an uptick in node count, which is then usually followed by a drop - although not in every case. This can be contributed to the increased activity from the community before the release. There could be a number of factors that cause this activity change: first, if the release date is fixed, the pressure on developers to include the planned features increases, which prompts more activity (pull model). Secondly, the rise in activity could be a newly implemented feature or a recently found bug, which needs immediate attention from developers. In this case it is possible, that the increased activity and larger volume of edits prompts a new release version (push model). Our assumption regarding whether a release follows a push or a pull model is that it can be highly dependent on the project, and within each project there could be a mixture of releases following both models. We cannot observe this trend for the patch releases, and we cannot draw a definitive conclusin regarding major releases, as there was only one major release among all the repositories subject to our analysis.

Regarding the layoff events marked with red vertical lines, a significan drop in the number of nodes can be observed shortly before the event in all projects. After the companies stop supporting the project and lay off the development team or they are reassigned to other topics, the number of nodes within the projects drop and stay low, never reaching the same numbers as

before. In case of `servo` and `wasmtime`, the first wave of layoffs do not seem to have a significant impact on the number of nodes, however, we do not have reliable information regarding how much the Rust development team was impacted by the first wave, on which both projects depend.

5.3.2 Network density and mean degree

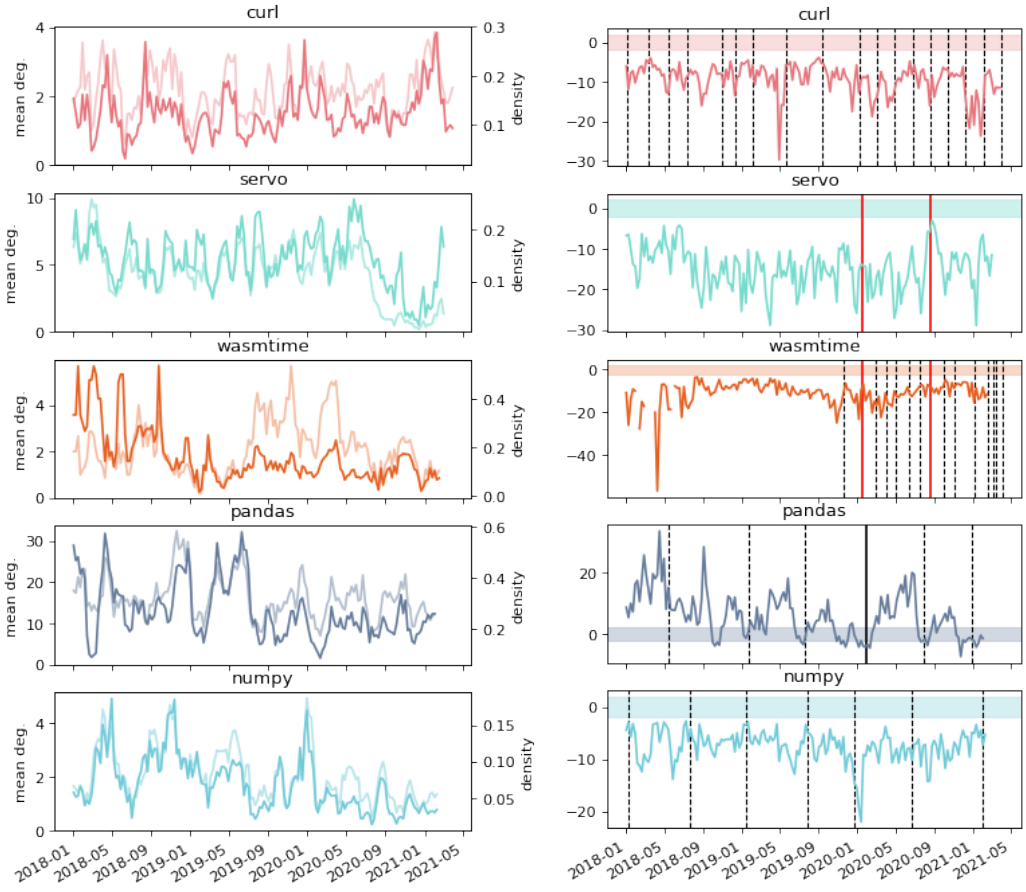
As a next step, we plot the network density and mean degrees of the projects. The network density tells us how densely connected is the network in a scale of $[0, 1]$, where 1 means every node is connected to all other nodes, and 0 means there are no edges in the graph. It refers to the ratio between all the possible connections and the actual connections. Although the generated networks are weighted, we do not consider the edge weight when counting each node's degree, meaning every connection is counted. We expect this value to rise if developers need to reach out to more members of the community and collaborate on more files with others. Similarly to the density, the mean degree value also only takes the number of nodes (N) and the degree number (k) of each node as inputs, so we expect similar results between the two measures.

By plotting the two measures on the same figure in Figure 12a, we can confirm, that they are closely correlated. The values are also highly sensitive regarding the number of nudes within the graph, and a low node count makes the graph values change radically. This is why the smaller projects are not shown, as their graph did not convey any meaningful observations. We can identify some troughs at the release times of minor releases, however, these could also be the results of the changes in the number of nodes.

In order to adjust the network statistics for their sizes, we compare the network with a randomly generated, same-sized network. If we see a significant difference between the two values, we can confidently state that the changes we observe in the network statistics are independent from the network size. To measure the significance, we calculate the *Z-value* of the actual network statistic and a 'general' network of the same size. In our implementation, we randomly generate 10 networks with the same number of edges and nodes as the original network, and use them to calculate the Z-value. The Z-value's formula is:

$$Z = \frac{x - \mu}{\sigma}$$

where x is the actual value we are comparing (in this case the network density), μ is the average network density of the randomly generated networks, and σ is the standard deviation of the random graphs. We consider a



(a) Density (dark) and mean degree (light). (b) Z-values for density and releases.

Figure 12: Network density, mean degree and Z-values over time with significance threshold highlighted.

network significantly different than a random network if its Z-value is greater than 2 in absolute terms. When the Z-value of network density is lower than -2, the network has a lower density than a same-sized general network, meaning the edges are more dispersed. Similarly, a Z-value greater than 2 signals that the graph has more central hubs than the random network.

We plot the Z-values over time in Figure 12b, where the area between -2 and 2 are colored to show at which times the network is significantly different than a randomly generated one: when the plot line is within this range, it is not significantly different, otherwise it is. In some cases, there is a break of continuity within the plot. This occurs, when the standard deviation of the randomly generated network is 0, which would result in a zero-division error. The standard deviation can only be 0 when all the values are the same, which means in these cases our randomly generated networks have exactly the same density. This usually happens in extreme cases, when there is a very low number of nodes or edges.

Almost all projects have a significantly low network density, meaning that the edges are more distributed between the nodes. The notable exception is the `pandas` repository, which has an overall positive Z-value over time, with sometimes crossing the insignificant zone. These points are mostly troughs at the time of minor releases. We can also see on all projects, that around a release, the Z-value approaches (or crosses) the threshold, which signals that the networks behave more like random networks around releases. The same can be observed for the layoff events, as `servo` gets really close to -2 at the time of the announcement.

5.3.3 Clustering coefficient

The weighted global clustering coefficients explained in Section 4.5.3 is plotted over time in Figure 13. Because it would be difficult to show both the Z-values and the the clustering coefficients on the same graph for each project, we only show the Z-values, as they both indicate the direction and the significance of the change. Where the clustering coefficient is significantly different from a random network to the negative side (meaning the Z-value at the corresponding date is less than -2) it drops below the highlighted $[-2, 2]$ area within each subgraph. Likewise, when there is a significant difference to the positive direction, the plot is above the highlighted area.

We receive mixed results for the clustering coefficients. The smaller libraries (`networkx`, `seaborn` and `curl`) show a sudden spike before a release, which means that the network gets clustered before a release. However, the Z-value is always showing a negative sign being less than -2 in these cases (with the notable exception of `pandas`, which is mainly above 2), signaling

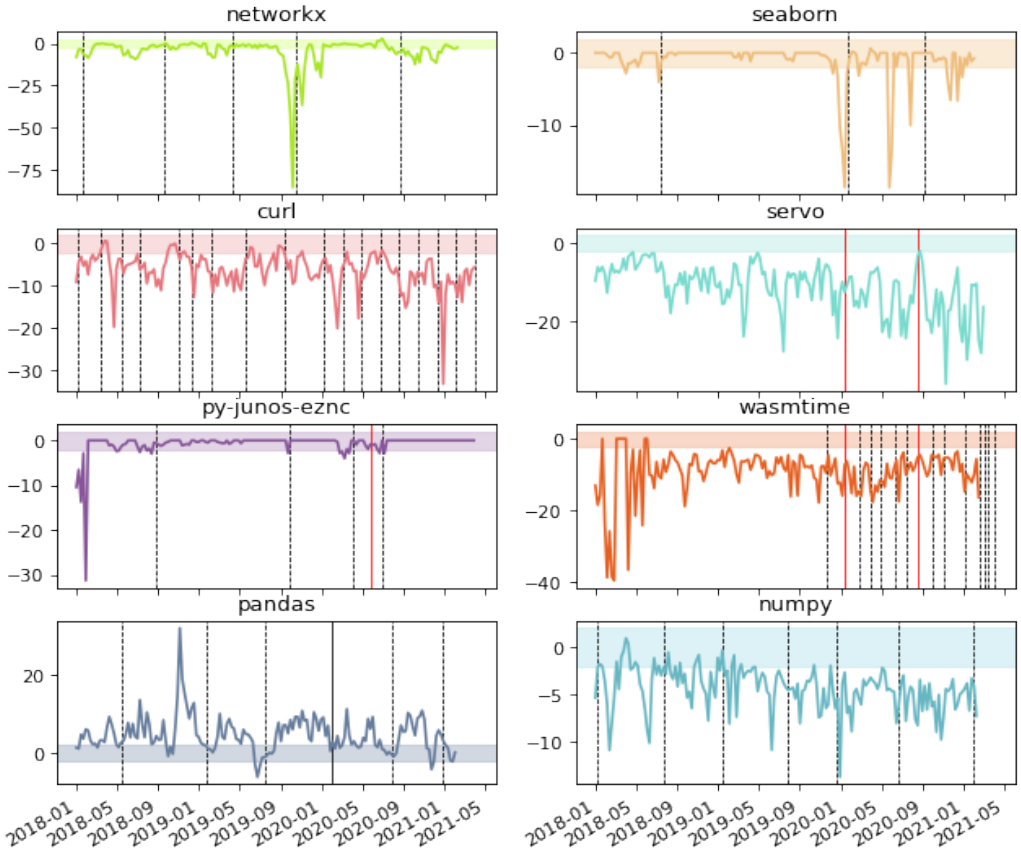


Figure 13: Clustering coefficient Z-values over time.

that the increase in clustering coefficient is significanly less than a same-sized random network. This shows, that the spike and then a drop in the clustering coefficient before a release is caused by a sudden increase and then decrease of activity within the project, but this activity is decentralized and affects all developers roughly the same. One of the reasons is the more decentralized workflow: as Crowston et. al. [10] have identified, smaller projects early in their lifecycles tend to be more centralized, as there is less specialization to only one area of the project.

Interestingly, almost all of the larger projects have consistently lower clustering coefficient than of a random network, which indicates, that there is a significant overlap of the edited files within the projects, and there is less specialization. Another explanation could also be the use of shared files, which relates to the project structure. If an implementation of a new feature requires developers to edit almost all existing files by adding new functions, then the clustering coefficient will be low. On the other hand, if every new feature requires a new file to be added, developers will be mostly isoated in the collaboration network around the files, which will show up as a higher clustering coefficient than in a random network. This would explain why the `pandas` library has consistently a positive Z-value: the project structure does not require contributors to edit eachother's files. Additionally, we can observe a drop in the clustering around releases, but without statistical significance. This also confirms, that the network becomes more random-like around releases, but we can further extrapolate that a likely factor to this phenomenon is the feature integration: between releases, contributors center around their specialization editing only a limited number of isolated files, but before the release, the effort shifts towards integrating these features together, which requires editing files outside of the 'usual folder'.

5.3.4 Mean path length

Another measure that could be affected by a releaase is the length between nodes in the network. From an organizational perspective, a path between two contributors represent the information flow regarding a specific part of the project. If in a network every vertex tends to be close to the other nodes, meaning less jumps are required, then they are able to gather information about a specific part of the project quicker. Smaller projects with less contributors naturally have relativel short connections to all other developers, and as the software gets larger with more contributors, the average shortest path between nodes is expected to get longer. However, if the project structure maintains central connecting 'hubs' (in this case: developers), the growth in length can remain lower.

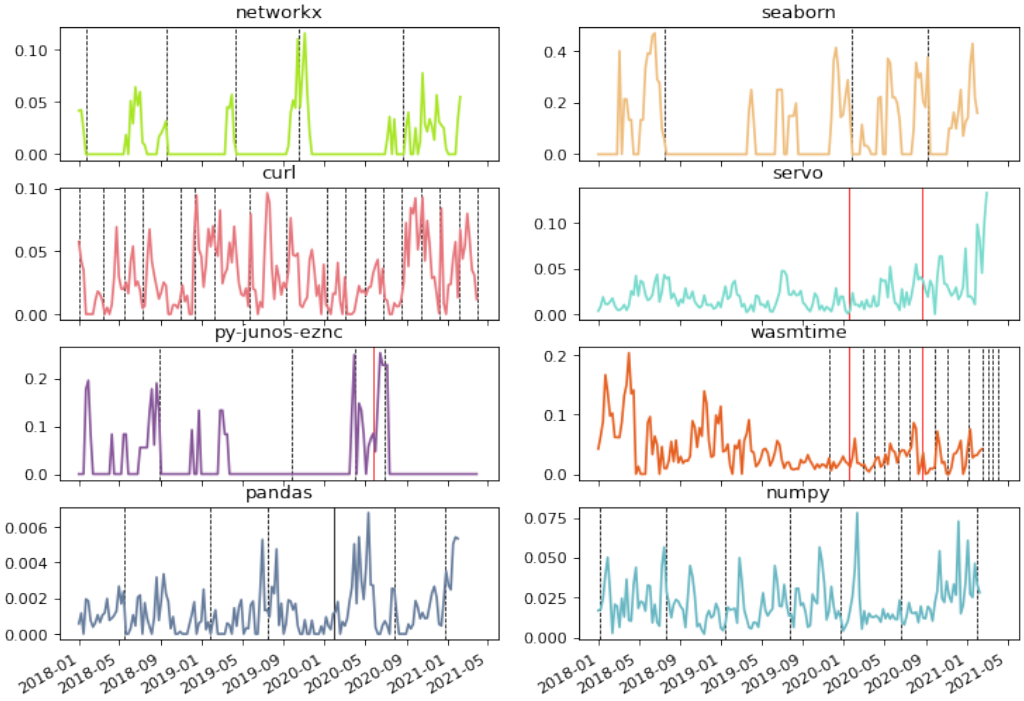


Figure 14: Normalized mean longest path length over time.

In an undirected unweighted graph, the shortest path between two nodes is the least amount of edges needed to be touched in order to get from one node to the other. However, our collaboration network is a weighted undirected graph, where the edge weights represent the collaboration effort between two developers. A strong collaboration is represented as a higher edge weight, whereas a lower edge weight means there is less collaboration between the two contributors. The shortest path between two nodes in a weighted graph is the path, which has the least summed weight of the edges touched. In our case, we want to measure how far away each developer are from all the others in order to quantify the effort for reaching out to eachother in case they want to involve a specific community member. To select the closest connection, we need to look for the strongest connection, meaning we have to consider the longest path between nodes.

We calculate the average longest path P by first summing up all the longest paths from a given node to all other nodes, then dividing by the number of connections, which is $n-1$ in a connected graph. Then we sum up all of these averages for all nodes, and divide by the number of nodes n:

$$P = \frac{\sum_{i=1}^{n} \sum_{j=1}^{n} - w_{ij} + 1}{n(n-1)}$$

where w_{ij} is the edge weight between a_i and a_j in range $[0, 1]$. We multiply the weights by -1 and add 1 in order to calculate for the longest, and not the shortest path. If there is no edge between a_i and a_j, then $w_{ij} = 0$.

To depict the changes over time in mean path length, we want to calculate the Z-value, just like with the other metrics. However, this proves to be difficult for the mean path length. When calculating the Z-value, we generate multiple graphs randomly, which have as many nodes and edges as the original, real network. Then we substract the original value from the average value of the random graphs, and divide with the standard deviation of the random network values. The issue rises with the standard deviation and division part: if all the randomly generated values happen to be exactly the same, their standard deviation will be 0, which leads to a division by 0 error. With the mean path, different random values can only happen if there are about the same edges as nodes in the network. In this case, random chains occur within the network, which are sometimes shorter, sometimes longer. Otherwise either everyone is connected to everyone and the mean path lenght is always 1 when there are more edges than vertices, or nobody is connected to anyone and there are never any changes in the reversed situation. In either case, Z-values cannot be calculated, as all values at all points in time are the same.

This occurs for all 8 projects when calculating the Z-value, and the graphs did not show any value. To circumvent this issue, we take an analytical approach to normalize our mean path length values. Instead of randomly generating the networks, we take the absolute possible largest mean path length in any network, and divide each value with it. The maximum path length in any given graph is $(N-1)/2$, where N is the number of developers in the network. We normalize the mean path length by dividing with this value, and the results are depicted in Figure 14.

It is clearly visible in the smaller projects (`networkx`, `py-junos-eznc`, `seaborn`), that the path length increases before a release due to the increased activity, and otherwise it stays 0. This indicates a 'push-type' release process, where the releases are sporadic, and a new release is created to incorporate and deploy the changes implemented. With the other projects, we do not see any correlation between releases and mean path length.

5.3.5 Core and periphery analysis

Around releases it is reasonably expected that the ratio between core and periphery developers changes. It has already been established, that within a project, the core members tend to stay the same throughout the project's lifecycle, but the tasks regarding the new release preparation might change

their relation with the periphery community members. The first method we use to identify core members is explained in Section 4.5.1. We call the number of core members identified this way as *Degree centrality core* value.

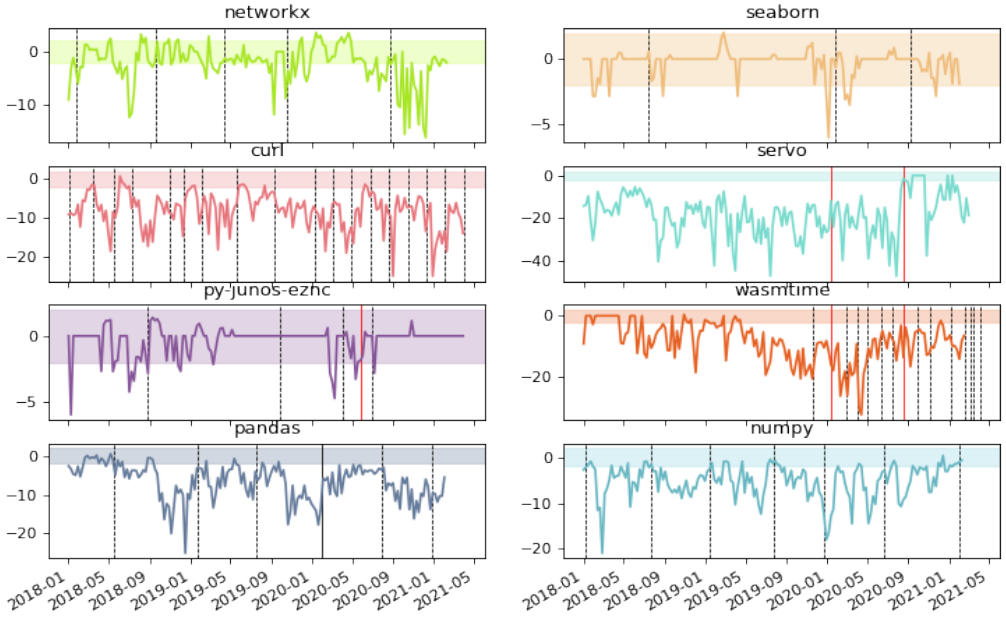
Besides taking the 20th percentile of the top clustering coefficients to identify the number of core developers in the network, we also use the degree number of each developer to identify their *K-core value*. The K-core of a network is the subset of nodes, which have at least k degrees, and we call the network's K-core value the number of nodes within the K-core [5]. Then, we identify the vertices, which are in the top 20th percentile of K-cores, as core developers. For example, if the maximum number of degrees within the network is 10, then the K-core value is the number of nodes, which have at least a degree number of 8.

We observe the ratio between the number of core and periphery developers by dividing the number of core developers with the number of nodes within the network. This gives us a ratio between 0 and 1 for both the *Degree centrality core* and the *K-core value* methods, where 1 means the network only contains core developers. It is important to note that the only case both methods can have 0 as a core/periphery ratio is when there is no collaboration within the network and all nodes are isolated with a degree of 0. We then plot the Z-values for both methods' core/developer ratio as before, which can be seen in Figure 15.

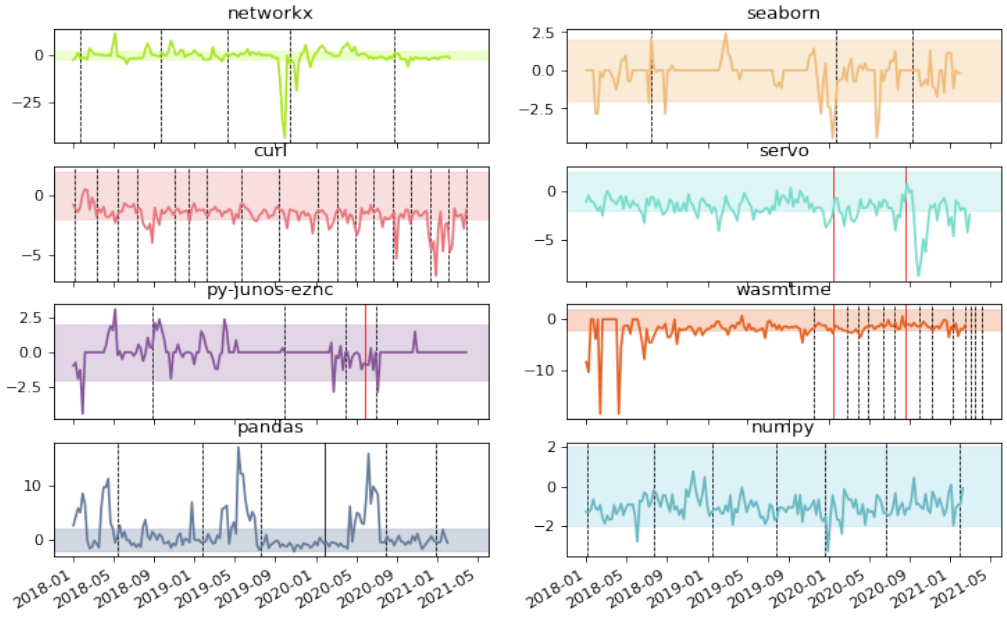
Figure 15a depicts the core/periphery ratio calculated with the K-core method, and Figure 15b shows the degree centrality method. We can see again a sharp difference between the smaller and larger projects. For the smaller projects, the two measures roughly behave the same, as their plots overlap for most of the time. Most of the time these values stay within the statistically insignificant range and only in these cases are significantly different than a random network.

In contrast, the large projects have their K-core ratios consistently below -2, indicating that the core ratio is consistently lower than a randomly generated network, which signals its centralized property. This cannot be said for the Degree centrality core ratios, which are consistently within the insignificant range. Just as with the smaller projects like `networkx`, `seaborn` and `py-junos-eznc`, we can observe a significant drop in K-core ratio Z-values within 2 to 3 months after the release. The difference between the two networks are demonstrated for `numpy`'s minor release on 22.12.2019, where the networks for 1-month after, and 1-month before are contrasted in Figure 16.

We can see the nodes identified as core highlighted with blue. In Figure 16a, we can see that the maximum number of degree is paired with a couple of similarly connected nodes, and this is combined with a relatively small

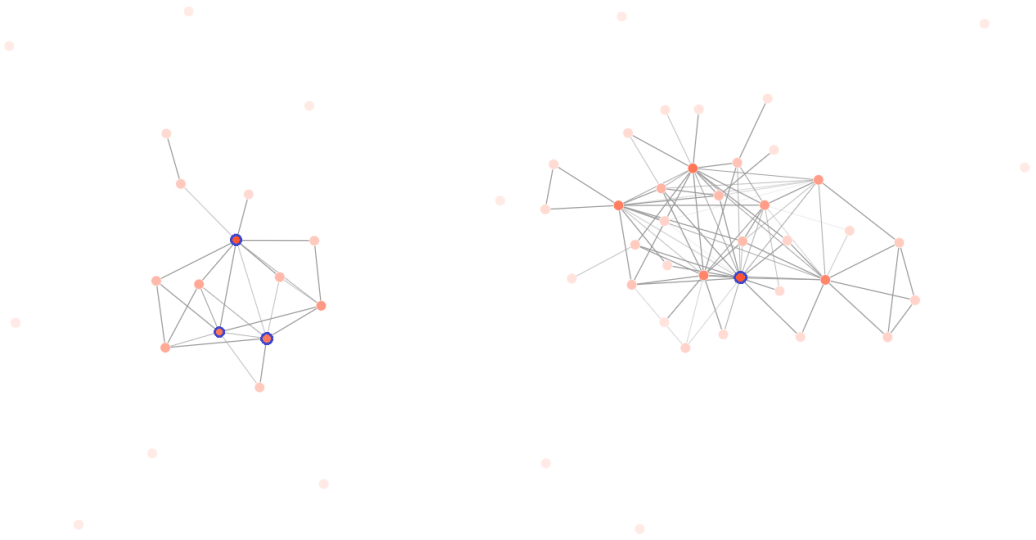


(a) Core/periphery K-core method



(b) Core/periphery degree centrality method

Figure 15: Core/periphery ratio's Z-value over time with K-core and Degree centrality methods.



(a) Between 2019-12-09 and 2020-01-06 (b) Between 2020-01-06 and 2020-02-03

Figure 16: `Numpy` collaboration network before and after the 2020-12-22 minor release.

number of nodes in the graph, which results in a relatively high number of core members with a low number of periphery, therefore the ratio gets higher, which is more similar to an almost completely dispersed random network. Whereas in Figure 16b, we can observe the opposite: the small number of core member (only one) is combined with a large number of nodes, which creates a low core-periphery ratio, that is significanly lower than in a randomly generated same-sized network. The fact that the second graph has more nodes and they are more connected lets us draw the conclusion that the increase in activity is the effect of the release, where after the new version release to the public led to an increase in bug reports, which caused a surge in activity as they are being fixed.

The notable exception for the K-core values in Figure 15 is again `pandas`, where the Degree centrality core-periphery ratio determines each release nicely: we can consistently see a positive spike above the significance level 2 months before each release. This finidng is consistent with the degree centrality analysis, however, here it is more pronounced. The drop after the release in K-core ratio also seems to hold true.

In general we can say that especially in larger projects the core can be clearly separated from the periphery developers, as the Z-values show significantly lower values than of a random network regarding the core/periphery ratio. We can also observe in projects with more contributors, that a release

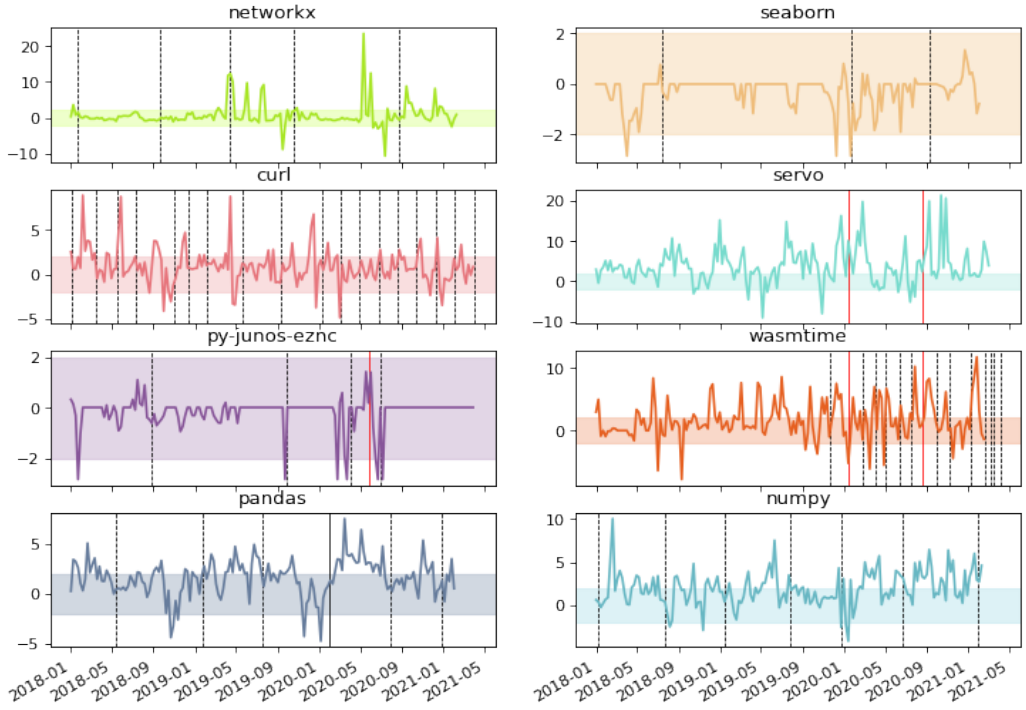


Figure 17: Hierarchy over time

causes the Z-values to rise, which indicates that the graph becomes more even. We can conclude from this observation, that more periphery contributors are active around a release, confirming the theory that an upcoming release triggers more activity as developers want to push their changes to the upcoming release.

5.3.6 Hierarchy

As a last step, we take a look at the hierarchy of the networks over time. We use the method described in Section 4.5.4 and we plot the linear regression's β_1 value (the slope of the trendline). Figure 17 shows these values over time per observed project, where the highlighted area indicates the zone where the Z-value is not significant.

In `networkx` and `seaborn` we can see activity only around releases, which is due to the fact that there aren't enough commits and developer activity to properly form a collaboration network, so it defaults to zero. Therefore, we can contribute the hiererchy change in their cases to the change in network size.

Just as with the other statistics, `curl` has a frequent and regular release

plan, which makes it hard to observe any regularities around releases. We can see some spikes and some troughs coincidentally at the same time as the releases, but there isn't any consistency between them, and therefore we cannot conclude a definitive cause and effect between hierarchy and releases. Unlike the other projects, `curl` has a lower hierarchy score as it drops below 0 many times, indicating a somewhat more hierarchial structure. This is expected, as it is known that it is mainly developed by a single developer, which results in high hierarchy.

In case of `servo`, `py-junos-eznc` and `wasmtime` the hierarchy seems to correlate with the unexpected events closely. In `servo` there is a definitive spike exactly at both after the first layoff event and exactly at the second layoff event, which clearly indicates a huge change in the network. Similarly, `py-junos-eznc` also shows the only significant Z-value at the time of the layoff event with a positive spike. `wasmtime` also shows a positive spike at the exact time of the layoff, which leads us to conclude that the sudden decline of hierarchy (it is more hierarchial if the value is negative) is a good indicator of the layoff events. This can be explained by the removal of core contributors, as the teams are disrupted and developers have to collaborate directly with more of their peers, as opposed to going through the established channels.

5.4 Release regularity

During the network measure analysis we recognized that the regularity of the releases can have a great effect on how it changes the network. In smaller projects a push-model can be recognised, where the new features are added, and then a new release is created, which incorporates this feature. This ensures that the new feature gets deployed as fast as possible. In these cases we can see that the releases do not happen regularly, but when there is a release, there is a significant change before or after it.

Projects that follow a pull-model tend to have more regular release periods, and they are also the large projects in our analysis. Within these projects the flow of new features and bug fixes are constant, and they are deployed within a fixed period of time. Therefore, the new release does not necessarily have a visible impact on the network, because if the feature is not complete yet, it will be released in the next one. A great example for this is the *1.0* release of `pandas`, which does not show any more network change in any metrics compared to a minor release despite being the only major release within all eight projects - contrary to the expectations.

In order to quantify the regularity of the releases within the project, we calculate the *coefficient of variation* (CV) within each project for the days

Project name	Minor+Major releases				Patch releases			
	N	σ	μ	CV	N	σ	μ	CV
`pandas`	6	34.93	191.2	**0.1827**	21	47.72	48.15	**0.9911**
`numpy`	7	22.91	186.67	**0.1227**	29	22.44	40.39	**0.5554**
`networkx`	5	46.70	235.75	**0.1981**	1	-	-	**-**
`seaborn`	3	164.5	392.5	**0.4191**	3	71.5	166.5	**0.4294**
`curl`	18	24.19	69.53	**0.3479**	8	99.29	136.0	**0.7301**
`servo`	116	8.31	8.67	**0.9585**	189	4.99	5.05	**0.9876**
`wasmtime`	13	24.49	41.91	**0.5843**	1	-	-	**-**
`py-junos-eznc`	4	127.68	224.33	**0.5692**	10	84.51	117.44	**0.7196**

Table 4: Release regularities for major, minor and patch releases.

between each release. The CV_i is the ratio of the standard deviation σ to the mean μ of number of days passed between two releases for project i, which can also be interpreted as a percentage [12]:

$$CV_i = \frac{\sigma_i}{\mu_i}$$

It must be determined for each type of release, because we have seen, that the type can have an effect of regularity. For example, in some projects patches follow up on a minor or major release, which can make the overall releases irregular, even though major and minor releases could still follow a regular pattern. We measure the release regularity CV in Table 4. In order to calculate the value, the project must have at least two releases per type to have at least one value for days passed between releases. In cases when this condition is not met, we leave the values empty.

The coefficient of variation has a lower bound of 0 in our case, because the number of days passed between two release can never be a negative number - the smallest possible is 0 when the release dates are exactly the same days apart from eachother. However, the CV has no upper bound, as the standard deviation can be arbitrarily large, and larger than the mean release days. Also, there is no unit associated with it, so we can only give meaning to the CV value in the context of other projects as comparison: a project with a smaller CV value has a more regular deployment schedule than another one with a higher CV. Since its scale is a ratio scale, we can also interpret a two times larger value as "two times more regular interval".

We also note that although the coefficient compensates for the difference between mean values, it does not take into consideration the number of intervals observed. This means that a one-day delay of a release from an established schedule does not have a large impact when there are 100 other releases, which are on time, but if there are only 4 releases, this changes the

mean significantly and therefore the CV too. This is why we also display the number of releases as N.

It can be observed in all projects, that the minor and major releases are more regular than the patch releases. This confirms the theory that patch releases are mainly added as bug fixes for minor and major releases after they are made public, and they do not folow a regular pattern. The difference between patches and major-minors are not consistent within the repositories: the greatest difference is in `pandas`, where the patch CV is five times larger than the minor-major CV. Close second is `numpy` with a similar 4.6 times larger irregularity of patches. Both projects have roughly a 180-day release cycle, and it is clearly visible that patches directly follow the minor releases in Figure 11, and they get sparser as we get further away from the minor releases in time.

The repositories `networkx` and `wasmtime` do not have enough patches within the observed period to calculate the patch values, therefore we cannot compare them to the major-minor releases. The `networkx` repo has a much more regular minor release cycle than `py-juno-eznc`, which has a similarly large release cycle of about 230 days. This indicates that small-sized projects, which usually have longer release cycles, can also vary a lot depending on the project, and they can be just as regular as a larger repository. `Seaborn` is somewhere in between the previous two in terms of regularity, however, with only 3 minor and 3 patch releases, we cannot say any definite about comparing it to other projects, because we cannot say for sure that the middle release of the 3 is an outlier of the many regular releases, or it is completely random even if we go back further in time to check a larger number of releases.

`Curl` has one of the shortest minor release cycles with an average of 70 days, and it is regular with a CV of 0.35, which confirms what we have observed visually on Figure 11. A short but regular release cycle will result in more added releases over a fixed time period, which leaves room for more errors. Furthermore, with a short release cycle only a small delay is required to have a great impact on the CV value. Therefore we believe that this method could be biased towards small release cycles, and they are shown as less regular than a larger, 180-day cycle shows. Nevertheless, it is still clear from the coefficient, that `curl` is one of the regular projects.

`Servo` has an extremely short release plan with only an average of 8-9 days between minor releases. We can also see the bias against shorter cycles, as it has by far the largest coefficient of variation with 0.96. Due to the short time between releases, it is enough to delay a release by just a couple of days to significantly worsen the CV score. However, with a standard deviation

also being around 8, we can see that the regularity cannot be as precise as in `pandas` or `numpy`, despite the bias.

Considering the plotted graphs with the releases, and also the CV values, we want to determine a CV value, that can classify a project into 'regular' or 'irregular' release cycles. We have seen that there is a slight bias against shorter cycles, but finding a way to compensate for it would greatly increase complexity, which could lead to other issues. Therefore, we will use a coefficient of variation of 0.4 **for the minor and major releases** to draw the line: every value below that indicates a regular release plan, and everything above that signals an irregular cycle. As stated before, patches tend to folow up on majors and minors, therefore we should not consider them overall. By this classification, `pandas`, `numpy`, `networkx` and `curl` are *regular*, whereas `seaborn`, `servo`, `wasmtime` and `py-junos-eznc` are *irregular* when it comes to release cycles. We can also see visually on the plotted graphs in Section 5.3, that this separation seems to be right.

5.5 Release efficiency, issues

So far we have seen how the collaboration network statistics change before, at and after a release. We also categorized the releases based on major, minor and patch releases, where we expected bigger changes in collaboration with a larger release - which was not always the case. Then we categorized the projects into regular and irregular release cycles. As a next step, we want to analyse in this section whether a release is successful or unsuccessful, and how we can measure this success rate.

The common perception of software development is, that a rushed release will have worse quality, which will result in more bugs in the software. This was confirmed by Michlmayr et. al. [27], and they name the lack of testing as one of the main source of rushed code. They also identify the two release strategies we mentioned in Section 5.4, namely regular, time-based or irregular, feature-based release strategies. They have found, that a feature-based strategy can cause rushed code development and poorly tested features due to the fact that the next release date is unknown. Therefore, contributors are incentivised to submit their changes as soon as possible, so that their changes are deployed to the public users of the open-source software. This unpredictability forces developers to skip or glance over the testing phases, which ultimately leads to more bugs in the software.

A fixed time-based release cycle, on the other hand, is predictable, which allows contributors to plan ahead the tasks related to the new feature or bug fixes [27]. Furthermore, in case of a delay, it can be already known when

the next one will be approximately, and further features can be replanned accordingly. A regular release schedule is therefore a sign of a healthy project, which indicates to the users, that there is active support for the OSS software, and also to developers that the project is set for a longer term, and it is a viable candidate to contribute into.

As a new version is released, the users of the software are notified through the distribution channels, which triggers a spike in downloads due to users wanting to be up-to-date to the latest software version [20]. Due to the quick adoption period within FLOSS projects, the majority of bugs are also expected to be reported shortly after a release, with a rushed version possibly having a larger amount of bugs after its release. Therefore, a great measure of release quality can be the number of bugs reported after a release. GitHub's standard bug and feature tracking API is the Issues page, so the number of issues created will be used for this purpose.

5.5.1 Issue opening and closing frequencies

The number of issues opened and closed can be seen on Figure 18. The issues are sampled in every 14 days, where the 7 days before and after are aggregated, separately for issues closed and issues created. For example, the month January will have two data points: January 8th, which aggregates the period January 1-14, and January 22nd, which will sum up all the number of issues created and closed between January 15-28.

Based on the plots, we can confirm the increased number of issues at a release, as all projects' most release dates coincide with a peak in issues created. Interestingly, the peaks do not necessarily come after or exactly at the release, and sometimes they preceed the actual release date. Some examples are the `pandas` version `v0.24.0` on *January 25, 2019* or version `v0.25.0` on *July 18, 2019*, or `py-junos-eznc`'s `2.3.2` on *April 1, 2020*. Most likely this is an effect of successful testing: before the release, the software is thoroughly tested, and all identified bugs are registered as issues. Since most issues are fixed before the release, users do not find more issues than usual, therefore the issues created drops. Similarly, a peak in opened issues after the release might mean a less robust testing system. We can observe this in `networkx`, where the number of issues created gets higher at the release, or shortly after.

The number of issues closed follows a similar trend. The expectation was, that the issues closed peak before a release, since the planned tasks to be added to the new version are being closed, and after the new release, the issues closed would fall, as most features and bugs have already been implemented in the new version. Although this holds true in some cases, especially in

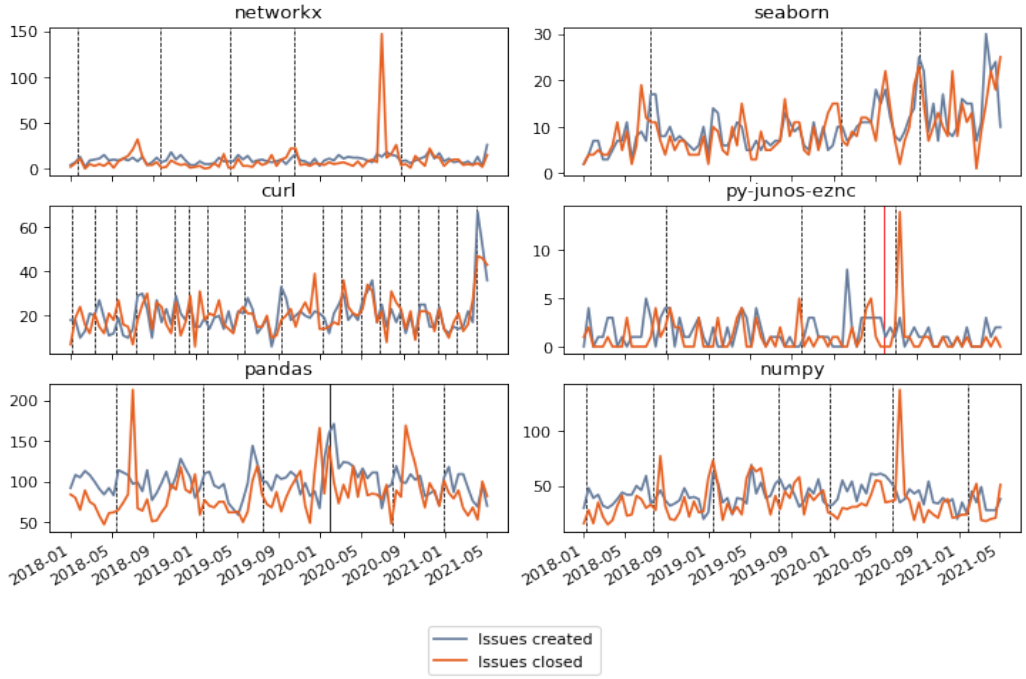


Figure 18: Issues closed and created per project.

`networkx` and `pandas`, this is not the case all the time, as sometimes large spikes occur right after a release, such as `numpy`'s version `v1.19.0` on *Jun 20, 2020* or `pandas`' new release on *May 15, 2018*. Our assumption is, that unusually large spikes like these occur as a 'clean-up' after the release, so the new release cycle can be planned.

5.5.2 Open issues over time

Now that we have seen some regularities regarding issue creation and close, we also examine how the number of open issues change with the releases. As we discussed in Section 4.6.2, since there are continuously open issues in an active project, some of which are not closed at the time of taking the project snapshots, we cannot take any averages or aggregation due to the survival bias.

Figure 19 shows the number of open issues for each project in the observed 3 years. We can see that in almost all projects, the number of open issues steadily rises, with smaller dips occurring around some, but not all releases. This shows, that maintaining and organizing the issues becomes a larger challenge as the project grows, and confirms the large spikes in the number of issues closed on Figure 18, that these are 'clean-up' events, where the

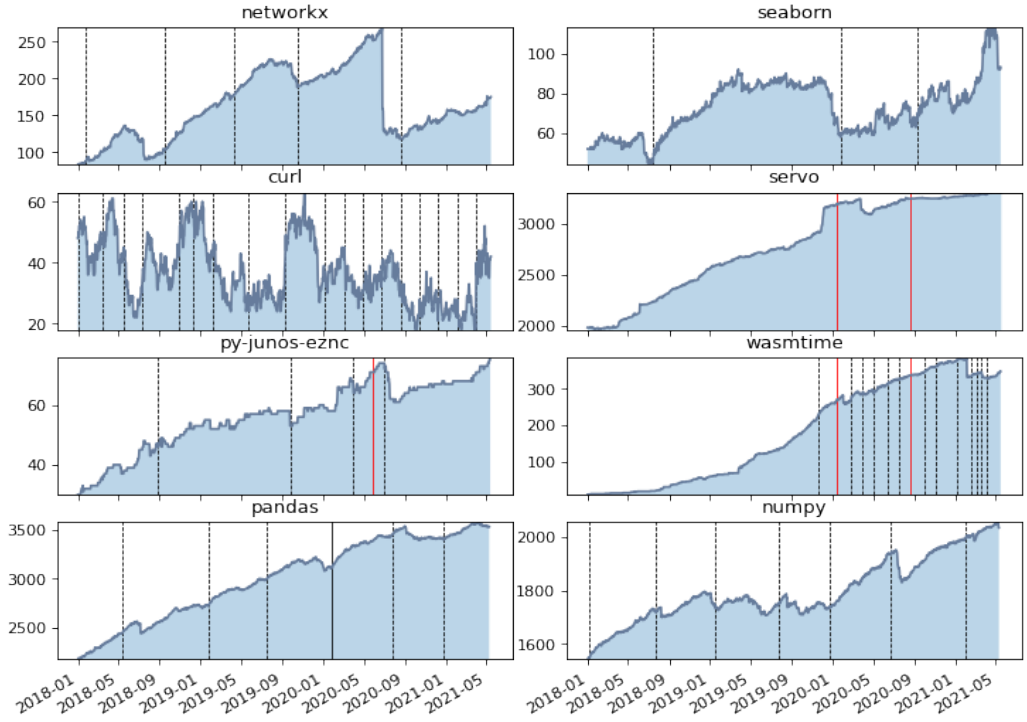


Figure 19: Open issues over time

backlog is decluttered by closing the obsolete issues.

The only exception from the ever-growing number of open issues is the `curl` library, which manages to keep the number of open issues below 60, despite being the oldest project of all 8 subject to our analysis. This could also be related to the short release cycle. As Khomh et. al. [20] have found, bugs are fixed significantly faster with rapid release plans. Furthermore, as `curl` is the most centralized project, with one single main developer, issue management is much more efficient, as there is no need for group discussions and prioritization between developers. Interestingly, `servo` has the shortest release timing strategy, with sometimes multiple minor releases in a week, yet it is still struggling with the increasing number of issues, indicating that a strongly centralized organizational structure can have a much heavier weight in the increase of open issues over time.

5.5.3 Bugs and features

Issues can describe many tasks within a project, including ideas, features, discussions, bugs, warnings, depreciation plans, and so on. Each repository organizes the issues with labels, which can be created and color-coded freely within each project. Therefore, the labels cannot be used across projects, as each repository has its own personalised system. For example, `pandas` uses the tag 'Bug' for bugs, `numpy` uses '00 - Bug', since they order the labels by giving a number in front of it, and `servo` doesn't have a bug category, because they divided this category into 'I-crash' and 'I-wrong' among other labels. Also, they categorize the labels by their starting letter, which is not a practice in the other projects.

Seeing which types of issues are affected the most by a release could also provide insight into how the project, and by extension the collaboration network changes. Instead of using the labels, we categorize them with pattern matching in the title as described in Section 4.6.2 with the same keywords and method. Even though the matching rate is not expected to be high, we can still apply the method for the projects, because the issues matching a category have a high chance of actually belonging to that category due to the low number of keywords being used, meaning our false positive rate for the classification is presumably low. However, we cannot confirm this, as the manual effort to classify the issues would be too large. The results can be seen in Figure 20.

The bugs and features are presented on two different scales, because their magnitude is different (there are a lot more bugs identified than features), but we want to compare the relative changes to eachother. Sadly we cannot

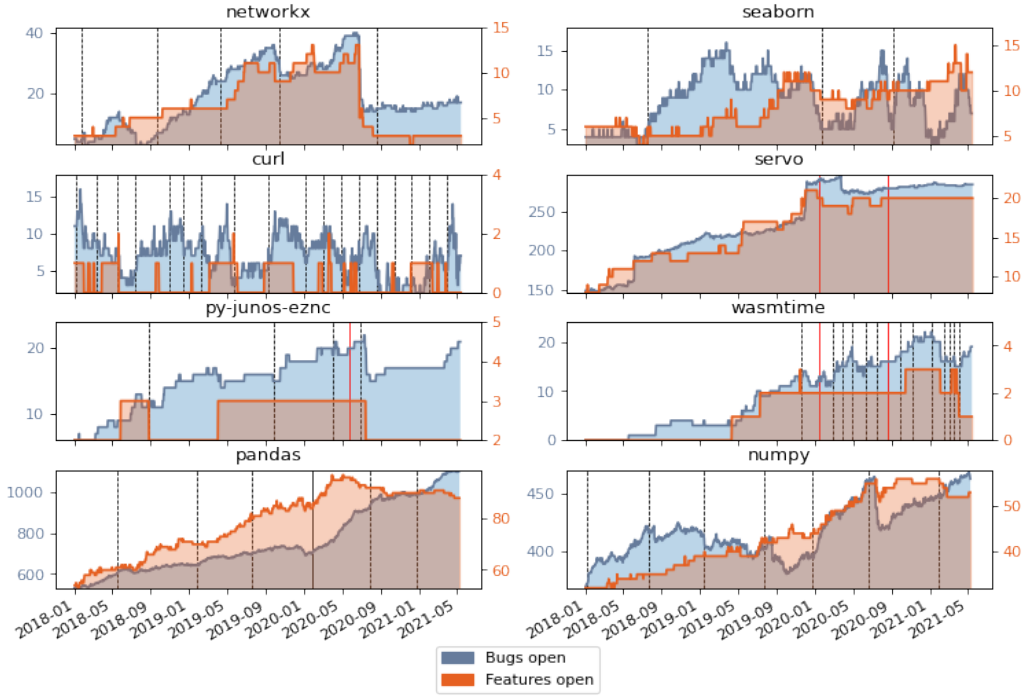


Figure 20: Open bugs and features over time.

see any significant difference between the number of bugs and features open, apart from a few exceptions, such as the December 2020 release of `pandas`, after which the number of open features decline while the number of bugs rise; or the July 2019 version of `numpy`, where the opposite is true and the features keep rising while the bugs decline.

Since both the features and the bugs curves follow the same shape, which is also the same as the overall shape of the issues, we cannot draw any definitive conclusions regarding the difference between bug and feature issues. The reason for this, is that we are only able to classify a small portion of the issues. When the overall number of issues rise, it makes it more likely to find a feature or bug just by sheer luck, and that is why both curves follow the overall curve. Therefore, we cannot confirm the statement that there is no difference either.

We summarize the classification results in Table 5. The vast majority of issues could not be classified, and on average only 23% can be identified as either bug or feature. Especially the feature recognition has the worst performance with a mere 1.36% of all issues on average. Although we believe that bugs should be more numerous within all repositories compared to features, our expectation is higher than the results we are seeing. We sug-

Project name	Issue count	Bugs		Features		Unclassified	
		N	%	N	%	N	%
`pandas`	20501	6245	*30.46%*	294	*1.43%*	13962	*68.10%*
`numpy`	9536	2421	*25.39%*	134	*1.41%*	6981	*73.21%*
`networkx`	2524	384	*15.21%*	54	*2.14%*	2086	*82.65%*
`seaborn`	1885	341	*18.09%*	99	*5.25%*	1445	*76.66%*
`curl`	2866	489	*17.06%*	35	*1.22%*	2342	*81.72%*
`servo`	11485	1016	*8.85%*	57	*0.50%*	10412	*90.66%*
`wasmtime`	917	72	*7.85%*	6	*0.65%*	839	*91.49%*
`py-junos-eznc`	475	106	*22.32%*	5	*1.05%*	364	*76.63%*
Overall	**50189**	**11074**	***22.06%***	**684**	***1.36%***	**38431**	***76.57%***

Table 5: Issue classification results.

gest for future research applying text mining techniques on not just the issue title, but also on the description. Additionally, duplicates could be filtered and sentiment analysis could be conducted on the issue title, description and received comments. However, these tasks are out of scope for the extent of our analysis.

5.6 Results

In our qualitative analysis we have identified the main metrics of a collaboration network, and observed these metrics in 8 hand-picked projects. Choosing a wide range of repositories in size, centralisation and release plans was a crucial criteria. We gathered each release of the projects over a 3-year period, and classified them as major, minor or patch releases based on its semantic versioning. Furthermore, we added the 'unexpected' events to this list for some selected repositories, as we also wanted to observe the layoff events and their effects on the network.

We looked at the number of commits, and we could identify that a project is active by the relative number of commits in a time period. For example, `servo` has significantly reduced number of commits after the second, bigger layoff wave. We also observed the number of nodes within the network rise in ceartain projects before a release, indicating an uptick in activity. In order to normalise for the network size and to compare each project to eachother, we introduced the Z-value, which tells us whether the measured value is significantly higher or lower than in a random network with the same number of edges and nodes.

The Z-values are consistently lower in almost all projects for network density, mean degree, clustering coefficient and mean path lengths, but the `pandas` library is an exception, as it tends to be positively different than a random network. This signals a stronger centralization and more organiza-

tional effort than in other repositories. Although `curl` is also very centralized as it is known that the main support comes from one developer, since everyone is centered around the main developer, it is not significantly different than other projects, where it is centered around a core developer network, thus implying `pandas` having multiple cores, which produce multiple central clusters within the network.

The number of overall contributors in a repository, which is a proxy for their size, also has a huge effect on the collaboration network. In smaller projects, activity tends to happen before a release, but in between releases or events even the 28-day time window proves to be too small to create a network which consists of more than just one or two developers working in isolation. Therefore, in these projects the network metrics are flat, and they only show activity before a release. In large projects, where there is a constant level of activity, the measures are inconclusive, as there isn't any single metrics, whose change consistently coincide with a major or minor release for all projects. Clustering coefficient and network density seem to work well in case of `pandas`, and the network density and the core/periphery ratio are also good predictors in `numpy`, but not for the rest.

Regarding the layoff events, hierarchy proved to be the best predictor, as all events coincided with a spike in hierarchy towards more decentralization. This is justified by the removal of the core members, as it naturally creates a more decentralized network. Also, hierarchy is a good predictor of releases in some cases, for example for some of the `pandas` releases, but not as significant as for the layoff events.

We have also seen that the repositories have varying release scheduling and strategies. Two parameters were identified: whether they follow a regular or irregular release schedule, and if there is a regular schedule, what is the time difference between two versions. We measured the regularity with the elapsed time periods' coefficient of variations, and identified the threshold of 0.4, below which a repository is considered as having a regular schedule, and above it is irregular.

Having a short release plan can obscure the changes in the network statistics, which could be one of the reasons why we cannot identify any metrics that predict well the new versions of `curl` and `wasmtime` libraries. Furthermore, according to the state-of-the-art literature, a regular deployment cycle makes the release plannign more predictable, therefore causes smaller changes in the collaboration metrics. We have found, that all of the large and medium projects (`servo, curl, wasmtime, pandas, numpy`) have adopted a regular release plan, while the small repositories have not. This leads to the conclusion, that switching to a regular release plan is a natural step in an OSS's lifecycle, and it is a sign of a healthy and mature project.

To discover the relationship between releases and issues, we observed the number of issues opened and closed with each new version. Our expectation was, that significantly more issues are closed before a release due to the bug fixes and features being included into the next version, and significantly more are opened after it because of the newly discovered bugs and feature ideas after a release is made public. We found this to be true for irregular release schedules, where the availability of a new feature prompts a new version, and not due to reaching a new release time. In repositories, that have adopted a regular release strategy and tend to be more mature, however, the opposite can be observed. Due to rigorous testing before a release date, issues are created before a new version is available, and they are closed within a clean-up process after the release is made publicly available.

Issue clean-ups usually happen around releases, but this is not necessarily the case in every project. When analysing the number of open tickets, we noticed, that in almost all projects, the number of open tickets grow over time. As an effort to keep the number of open issues manageable, sometimes project organizers close a significant amount of tickets at once, causing a spike in the number of closed issues. However, only `curl` seems to be able to keep the open issues at a constant level, which we contribute to having only a single developer as a core member, which eliminates the necessity to discuss, prioritize and triage open issues.

In order to confirm the theory, that the number of bugs increase after a release, we classified the issues into bugs and features based on keywords and pattern-matching. However, due to the heterogenous nature of issue categorization and labeling, we were only able to classify a small percentage of issues. This caused the number of bugs and features to be tied to the overall number of issues: when the number of open issues are increasing, we are more likely to find and classify a bug and a feature, so their numbers will also rise. This effect ties the number of both bugs and features to the overall number of issues, so their shapes look approximately the same over time, and we are not able to draw any conclusions from them. As further analysis is out of scope for our work, we suggest a more robust classification method for bugs and features with text mining tools, such as natural language processing (NLP) for not just the titles, but also the description and comments of the issues. Furthermore, a sentiment analysis on the comment responses could provide more insight into how issues are resolved.

6 Quantitative OSS project analysis

In Section 5 we have seen how the heterogenity of open source projects makes it hard to draw conclusions from the network statistics over time. We have seen some correlations between the releases and the metrics, but they weren't applicable to all repositories. Furthermore, the relatively small nubmer of repositories also prevented generalizing the findings, for example the metrics, which proved to be effective in `pandas` like clustering coefficient or degree centrality, might also apply to the majority of open-source repositories, but our project choices did not include any, that falls into this group. Therefore, in this section we will conduct a quantitative research on a large number of randomly selected repositories to rule out any bias we might have applied during the pattern analysis, where the main focus was to understand the collaboration network changes over time.

First, we query a large number of GitHub repositories. As we have seen in the previous chapter, the analysis works best on really large repositories, otherwise the activity tends to be zero between releases. Therefore, we filter for only the largest projects available on GitHub with specifying inclusion and exclusion criteria. As a next step, we randomly choose a number of projects from the result set, which we mine using git2net and repo_tools. Then we run a linear regression on the number of issues, the releases and the network statistics before and after a release to discover the correlations between these variables. Lastly, we discuss our findings and point to future research topics.

6.1 Inclusion and exclusion criteria

We use the *GitHub Search*[7] tool maintained by SEART to query the Github repositories. The tool allows us to set a detailed search criteria to find repositories. We use the following settings for querying:

- *Commits*: minimum 5000
- *Contributors*: minimum 5
- *Issues*: minimum 10
- *Pull requests*: minimum 10
- *Releases*: minimum 2

[7] https://seart-ghs.si.usi.ch/

- *Stars*: minimum 10
- *Watchers*: minimum 10
- *Forks*: minimum 2
- *Exclude forks*: True
- *Has open issues*: True
- *Has open pull requests*: True

The main selection criteria is the number of commits, as this measures the best how much work has been put into the project. All other variables might change in any direction over time, for example repos can be unwatched, unstarred, forks can be deleted or merged and contributors might decline over time. However, the number of commits always rises as time passes.

In our qualitative analysis and during its mining process we have seen, that the larger projects, which showed activity in a 28-day window regardless whether a release was coming up or not, had at least 5000 commits, this is why we used this number as a filter. The other criteria (contributors, issues, pull requests, etc...) are filtered, so that we are making sure that all of the GitHub functionalities are being used, e.g. issues are managed in GitHub, it has at least a minimum number of contributors so the networks will not be empty, and it has at least a small community due to having stars, watchers and being forked. We exclude forked results, as we are only interested in the main source code, and we mark the 'Has open issues' and 'Has open pull requests' to filter for active projects.

Applying the above inclusion and exclusion criteria, the search resulted in 2211 repositories in total. We further narrow this result set down to 110 by randomized selection. Our goal is to analyse about 100 repositories, but due to various mining and release problems, this number is expected to be further reduced, therefore we choose an extra 10 repositories to compensate the expected reduction.

Since following the semantic versioning standards is a crucial part of the analysis, we also exclude the projects, which do not follow the major-minor-patch versioning convention, or if they have multiple parallel editions, which are versioned separately. We have to exclude the latter category, as the version is determined based on the version number change compared to the previous version, and if the previous is denoting a different edition of the software, the version type can be misidentified. For example, the `chakra-ui/chakra-ui` repository contains releases named '**select@X.Y.Z**', '**props-docs@X.Y.Z**' and also '**progress@X.Y.Z**', where X, Y and Z are

integers. If the 'select@1.0.1' release is followed by the 'props-docs@1.1.6', then based on our version denoting algorithm, this would be categorized as a major release, even though the latter release is just a patch to 'props-docs@1.1.5', just chronologically it is not the preceeding release. Since exceptions and personalizations are common between repositories regaring versioning, we decided not to investigate further how these projects could be incorporated, as this would make the classifying algorithm significantly more complex. Instead, we propose an improved algorithm for future research.

6.2 Repository mining

The above exclusion criteria results in 84 repositories, which we start mining with both `git2net` and `repo_tools`. Just like in the qualitative analysis, the data regarding the commits and edits, which are used to generate the collaboration networks, are gathered by `git2net`, and the releases and issues details are collected with `repo_tools`. During the two-week long mining process, 6 repositories were skipped due to the commits and edits mining running into an error or getting stuck. Presumably this is due to certain files (possibly large binary files), which require too many resources to compare the changes line by line. Part of the mining process is the disambiguation of commit authors, as renamed users can show up multiple times within the collaboration network. The original author names are kept in `.csv` files for each mined repo.

As a last step in the mining process, we mine the repositories with the `repo_tools` miner. Of the remaining 78 projects, 10 could not be mined for various reasons, including the miner getting stuck with some of them during the commits mining or the repository being renamed and therefore does not match the provided URL anymore. We show the summary of the 78 projects in Table 6 and the main programming languages and their distribution in Table 7. We also included the 10 repositories skipped, because they finished for the `git2net` mining process, but we will exclude them during the analysis going forward, and we will only work with the fully successful 68 repos. The full list of the 110 randomized repositories can be found in Appendix A.

6.3 Metrics and release stats correlation

We investigate the correlation between the network metrics and the releases with linear regression. First, we compose a list of all the mined repositories and all of their releases with the release date. Then the release regularity is calculated for each repository using the coefficient of variance value descibed in Section 5.4. Since this value requires more than one release, the CV is

Attribute	Min	Avg	Max	St.dev.
Commits	5049	11768	54396	9168
Branches	1	47	408	67
Releases	2	95	889	150
Contributors	6	137	1181	179
Watchers	10	234	7120	816
Stargazers	11	5059	149037	17790
Forks	8	1637	72735	8259
Issues	11	2293	20235	3490
Pull requests	27	2205	22271	3319

Table 6: Basic statistics of mined repositories.

Main language	Count	%
C++	21	26,9%
JavaScript	16	20,5%
Python	13	16.7%
Java	13	16,7%
C	7	9,0%
TypeScript	4	5,1%
C#	3	3,8%
Kotlin	1	1,3%

Table 7: Main programming languages distribution amongst mined repositories.

measured per repository and not per each release. Lastly, we calculate all of the collaboration network metrics for the network containing the past 28-day activity before the release, and the network covering the 28 days right after the release.

6.3.1 Release classification

To calculate the CV value, the patch releases must be excluded as we mentioned in Section 5.4. First, we classify each release into major, minor or patch releases. Because the semantic version can only be determined by comparing a release with its preceeding version, the first release in each repository is dropped because its version will always be unknown as it lacks a previous release. Furthermore, this type of classification also has issues categorizing parallel version maintenance. For example, if a project parallelly supports a version v2 and a version v3, it issues new minor and patch releases for each version, so we will see `v2.1.4`, `v3.0.2`, `v2.1.5` and `v3.0.3` in chronological order. In this theoretical example, the releases `v3.0.2` and `v3.0.3` will be categorized as major releases, because their preceeding release has a lower major number. However, it is clear, that these should be patch releases, since they follow up on the previous patch for the respective version. In order to circumvent this issue, we would need to update the classifying algorithm to consider all the preceeding versions. However, this would significantly inclrease the complexity of the program, and therefore it is out of scope for our work. We mark a robust classification algorithm as a possible future topic to be researched.

Another problem with the version classification are the irregular release names, such as beta releases or release candidates. These versions usually have the same semantic version number, and they are only different in the optional tags at the end. As an example, during major releases it is common to see multiple release candidate versions following eachother, like `v3.9.12` followed by `v4.0.0rc1` and `v4.0.0rc2`. The issue rises from the fact that by looking at only the release number, the latter two releases are the same to the classifying algorithm (4.0.0), which leads to `v4.0.0rc1` being classified as a major release, and `v4.0.0rc2` will be unknown. We could mark these versions as rc versions, and skip them during the analysis, but the heterogenity of conventions is a great issue, as some projects mark release candidate, beta and alpha versions as `v4.0.0-rc1`, `v4.0.0beta5` or even subversioning them such as `v4.0.0-rc1.2`. Some projects also have different editions marked with an symbol, for example `v3.1.5@latest` and `v3.1.5@browser-only`, which also results in the same issue. To resolve all the releases, which are uncategorizable, we could only make it 100% accurate without any unknowns

Repo	Release	CV	Type	Change size	N (b)	...	Hierarchy (a)
nccgroup/ScoutSuite	4.0.4	0.5	patch	780726	3	...	0.00
nccgroup/ScoutSuite	4.0.5	0.5	patch	46	3	...	0.00
nccgroup/ScoutSuite	4.0.6	0.5	patch	70470	3	...	0.08
nccgroup/ScoutSuite	4.1.0	0.5	minor	2708046	8	...	-0.08
nccgroup/ScoutSuite	4.2.0	0.5	minor	85243130	8	...	-0.07
...	...	...	...	...	...	...	...
Unidata/netcdf-c	v4.7.1	0.7	patch	636254	3	...	0.00
Unidata/netcdf-c	v4.7.2	0.7	patch	411017	3	...	0.10
Unidata/netcdf-c	v4.7.3	0.7	patch	702122	8	...	0.00
Unidata/netcdf-c	v4.7.4	0.7	patch	60941386	5	...	0.10
Unidata/netcdf-c	v4.8.0	0.7	minor	129245816	6	...	0.00

Table 8: OLS regression input data. (b=before, a=after)

if we manually modified or deleted the release data. Provided, that the 68 mined repositories contain over 5200 distinct releases, this task would be too cumbersome, and therefore we marked all releases affected by this as 'unknown'. Because we would like to perform a linear regression on our dataset, we must convert all nominal categorical labels to numbers, thus we add 3 columns to the current dataset named 'major', 'minor' and 'patch', where these values will have 1 in their respective release type column, and the other two will be 0. All three columns are zero if the release type is unknown. We also keep the nominal category in case of models, which can handle categorical values.

Because the semantic version can be highly unreliable in certain repositories, we take a look at how much was changed between two releases to identify the significance of the release. We do this by taking all commits between two relositories, and adding up the number of lines added and number of lines deleted. Since this method only requires the preceeding version's release date, it provides a measure of significance without the classification errors mentioned above. However, it is also important to note that the version type (major, minor, patch) is not necessarily related to the number of lines changed. The resulting dataframe is depicted in Table 8.

6.3.2 Semantic version number and number of changed lines

First, we observe how the semantic version numbers predict the number of changed lines by setting the *major*, *minor* and *patch* columns as predictors and the *Change size* as the intecept (y value). Our null hypothesis is that the semantic version number and type does not have an effect on the number of lines changed in a new release compared to the previous version.

The OLS regression results with *patch* being the reference level can be seen in Table 9. We used the formula $log(change) \sim C(repo) + C(type)$,

Variable	coef	std err	t	P>|t|	[0.025	0.975]
major	0.87	0.2	4.0	0.0	0.43	1.3
minor	1.61	0.2	10.3	0.0	1.3	1.9

Table 9: OLS regression results for number of line changes with semantic version type as predictor.

where the *change* is the number of lines changed, *repo* is the repository's name and *type* is the semantic version type as string (major, minor or patch). The *log*() is used for the number of lines changed because a larger release is expected to have exponentially more lines changed, and the $C()$ signals the `statsmodels` package that the variable is a nominal variable. The resulting linear regression equation

$$F(61, 3417) = 30.87, p < .000$$

with $R^2 = .355$ is found to be significant, based on which we reject the null hypothesis. Surprisingly, the major version's coefficient is twice lower than the minor version's, which is contrary to our expectations. The number of lines changed were expected to be higher in a major release than in a minor, but this was found not to be true. We partially contribute this to the misclassification due to parallel versioning, since most projects support only major versions parallelly - if they do. Because of this, the most falsely labelled versions are major versions, which causes inconsistency when determining the number of lines changed.

6.3.3 Correlation between network metrics

In Section 5 we observed, that the collaboration network metrics can be dependent on eachother. In this section, we analyse their correlation by creating a matrix of correlations, which is depicted in Figure 21.

Correlation coefficients, whose value is between 0.9 and 1.0 are classified as very highly correlated. Values between magnitudes 0.7 and 0.9 are considered as highly correlated, between 0.5 and 0.7 the two variables are moderately correlated, and datasets with a correlation coefficient between 0.3 and 0.5 are categorized as having low correlation. Variables, whose correlation coefficient is between 0 and 0.3 do not have any significant correlation, as the two datasets are so different, that the occasional similarities are most likely caused by chance and no underlying correlation. These ranges are also applicable to values below 0 with a reversed sign, and the negative sign signaling an inverse relationship.

Based on these categories, the following correlation types can be seen:

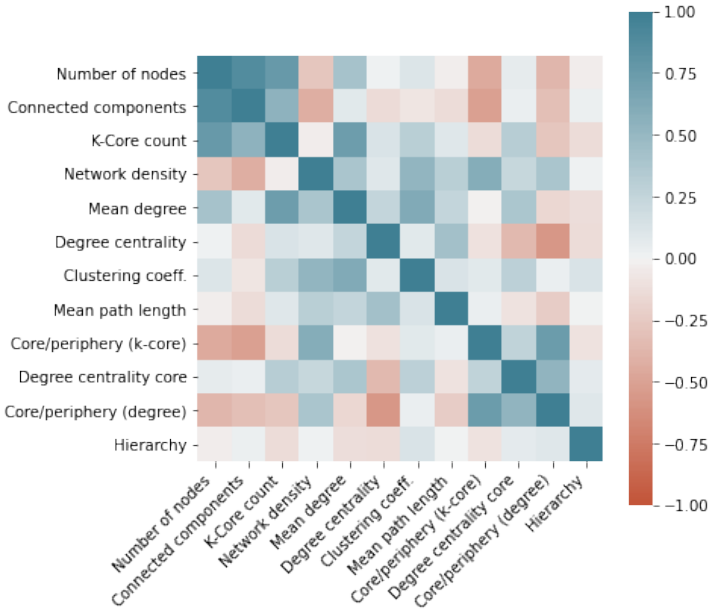


Figure 21: Correlation matrix of release regularity, lines changed and the other collaboration metrics before the release.

- **High positive linear correlation:**
  Number of nodes ∼ Connected components (0.88)
  Number of nodes ∼ K-core count (0.76)
  Core/periphery (k-core) ∼ Core/periphery (degree) (0.74)
  Mean degree ∼ K-core count (0.73)

- **Moderate positive linear correlation:**
  Mean degree ∼ Clustering coefficient (0.61)
  Network density ∼ Core/periphery (k-core) (0.59)
  Connected components ∼ K-core count (0.54)
  Degree centrality core ∼ Core/periphery (degree) (0.53)
  Network density ∼ Clustering coefficient (0.52)

- **Low positive linear correlation:**
  Degree centrality ∼ Mean path length (0.43)
  Number of nodes ∼ Mean degree (0.41)
  Network density ∼ Mean degree (0.39)
  Network density ∼ Core/periphery (degree) (0.39)
  Mean degree ∼ Degree centrality core (0.39)
  K-core count ∼ Degree centrality core (0.33)
  Clustering coefficient ∼ K-core count (0.31)

Network density ∼ Mean path length (0.30)

- **Low negative linear correlation:**
  Number of nodes ∼ Core/periphery (k-core) (-0.46)
  Network density ∼ Connected components (-0.44)
  Number of nodes ∼ Core/periphery (degree) (-0.37)
  Degree centrality ∼ Degree centrality core (-0.36)
  Connected components ∼ Core/periphery (degree) (-0.32)

- **Moderate negative linear correlation:**
  Degree centrality ∼ Core/periphery (degree) (-0.57)
  Connected components ∼ Core/periphery (k-core) (-0.51)

From these significant correlations and based on Figure 21 we can see that the number of nodes has one of the largest correlation with all the other variables. This rpoves, that one of the greatest influence on the network metrics is the number of developers, and in extension, the activity within the repository. The k-core count, which measures the number of nodes having a degree number within the top 20th percentile, is highly positively correlated with the number of nodes, which indicates, that the number of core developers in the network increases in proportion to the network size. However, the number of core developers classified by the degree centrality method does not indicate any relationship, which leads to the conclusion that these two methods do not identify the same collaborators as core members all the time, and there are significant differences. The low positive linear correlation between the two variables also confirms this observation.

The two derived core and periphery measures, however, are both negatively correlated with the number of nodes, indicating that as the network grows, the core becomes relatively smaller within the graph, meaning that most of the network growth can be contributed to the increased number of periphery developers. This observation is in line with the findings of McClean et. al. [26], who found that the number of core members tend not to change. The two core-periphery ratio metrics are also show a high positive correlation, which shows that they both cover the same concept and measure the same property of the network.

TODO...

6.3.4 Pre- and post-release network metrics and lines changed

TODO...

6.3.5 Pre- and post-release network metrics and regularity

TODO...

6.4 Project success

7 Discussion and results

8 Conclusion and future work

References

[1] Shaosong Ou Alexander Hars. Working for Free? Motivations for Participating in Open-Source Projects. *International Journal of Electronic Commerce*, 6(3):25–39, April 2002.

[2] Mohammed Aljemabi and Zhongjie Wang. Empirical Study on the Evolution of Developer Social Networks. *IEEE Access*, PP:1–1, September 2018.

[3] M. Antwerp. Evolution of open source software networks. In *OSS 2010 Doctoral Consortium, Collocated with the 6th International Conference on Open Source Systems, OSS 2010*, pages 25–39, January 2010.

[4] Guilherme Avelino, Leonardo Passos, Andre Hora, and Marco Tulio Valente. A Novel Approach for Estimating Truck Factors. *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10, May 2016.

[5] V. Batagelj and M. Zaversnik. An O(m) Algorithm for Cores Decomposition of Networks. *arXiv:cs/0310049*, October 2003.

[6] Christian Bird, David Pattison, Raissa D'Souza, Vladimir Filkov, and Premkumar Devanbu. Latent social structure in open source projects. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 24–35, New York, NY, USA, November 2008. Association for Computing Machinery.

[7] VM (Vicky) Brasseur. Farewell, Juniper Networks, June 2020.

[8] Melvin E Conway. How Do Committees Invent? *Datamation*, 14(4):4, 1968.

[9] Kevin Crowston and James Howison. The social structure of free and open source software development. https://firstmonday.org/ojs/index.php/fm/article/download/1478/1393, February 2005.

[10] Kevin Crowston and James Howison. Hierarchy and centralization in free and open source software team communications. *Knowledge, Technology & Policy*, 18(4):65–85, December 2006.

[11] Ikram El Asri, Noureddine Kerzazi, Lamia Benhiba, and Mohammed Janati. From Periphery to Core: A Temporal Analysis of GitHub Contributors' Collaboration Network. In Luis M. Camarinha-Matos, Hamideh Afsarmanesh, and Rosanna Fornasiero, editors, *Collaboration in a Data-Rich World*, IFIP Advances in Information and Communication Technology, pages 217–229, Cham, 2017. Springer International Publishing.

[12] Brian Everitt. *The Cambridge Dictionary of Statistics*. Cambridge University Press, Cambridge, UK, New York, 1998.

[13] Christoph Gote, Ingo Scholtes, and Frank Schweitzer. Analysing Time-Stamped Co-Editing Networks in Software Development Teams using git2net. *arXiv:1911.09484 [physics]*, November 2019.

[14] Christoph Gote and Christian Zingg. Gambit – An Open Source Name Disambiguation Tool for Version Control Systems. *arXiv:2103.05666 [physics]*, March 2021.

[15] Oskar Jarczyk, Szymon Jaroszewicz, Adam Wierzbicki, Kamil Pawlak, and Michal Jankowski-Lorek. Surgical teams on GitHub: Modeling performance of GitHub project development processes. *Information and Software Technology*, 100:32–46, August 2018.

[16] Arkadiusz Jędrzejewski. *The Role of Complex Networks in Agent-Based Computational Economics*. PhD thesis, Wroclaw University of Technology, June 2016.

[17] Mitchell Joblin, Sven Apel, Claus Hunsen, and Wolfgang Mauerer. Classifying Developers into Core and Peripheral: An Empirical Study on Count and Network Metrics. *arXiv:1604.00830 [cs]*, April 2016.

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

[19] Jacob Kastrenakes. Mozilla is laying off 250 people and planning a 'new focus' on making money. https://www.theverge.com/2020/8/11/21363424/mozilla-layoffs-quarter-staff-250-people-new-revenue-focus, August 2020.

[20] Foutse Khomh, Tejinder Dhaliwal, Ying Zou, and Bram Adams. Do faster releases improve software quality? An empirical case study of Mozilla Firefox. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 179–188, June 2012.

[21] Frederic Lardinois. Mozilla lays off 70 as it waits for new products to generate revenue.

[22] Yu-Ru Lin, Hari Sundaram, Yun Chi, Junichi Tatemura, and Belle Tseng. Blog Community Discovery and Evolution Based on Mutual Awareness Expansion. In *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence*, pages 48–56, November 2007.

[23] mariogrip. Semantic versioning for UT. https://forums.ubports.com/topic/1822/semantic-versioning-for-ut, October 2018.

[24] Juan Martinez-Romo, Gregorio Robles, Jesus M. Gonzalez-Barahona, and Miguel Ortuño-Perez. Using Social Network Analysis Techniques to Study Collaboration between a FLOSS Community and a Company. In Barbara Russo, Ernesto Damiani, Scott Hissam, Björn Lundell, and Giancarlo Succi, editors, *Open Source Development, Communities and Quality*, volume 275, pages 171–186. Springer US, Boston, MA, 2008.

[25] M. R. Martínez-Torres. A genetic search of patterns of behaviour in OSS communities. *Expert Systems with Applications*, 39(18):13182–13192, December 2012.

[26] Kelvin McClean, Des Greer, and Anna Jurek-Loughrey. Social network analysis of open source software: A review and categorisation. *Information and Software Technology*, 130:106442, February 2021.

[27] Martin Michlmayr, Brian Fitzgerald, and Klaas-Jan Stol. Why and How Should Open Source Projects Adopt Time-Based Releases? *IEEE Software*, 32(2):55–63, March 2015.

[28] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM*

Transactions on Software Engineering and Methodology, 11(3):309–346, July 2002.

[29] Rick Olson. Release Your Software, July 2013.

[30] Jukka-Pekka Onnela, Jari Saramäki, János Kertész, and Kimmo Kaski. Intensity and coherence of motifs in weighted complex networks. *Physical Review E*, 71(6):065103, June 2005.

[31] Tom Preston-Werner. Semantic Versioning 2.0.0. https://semver.org/.

[32] PwC. Leading benefits of open-source software among enterprises worldwide as of 2016. *Statista*, 2016.

[33] Erzsébet Ravasz and Albert-László Barabási. Hierarchical organization in complex networks. *Physical Review E*, 67(2):026112, February 2003.

[34] Rotem Stram, Pascal Reuss, and Klaus-Dieter Althoff. Weighted One Mode Projection of a Bipartite Graph as a Local Similarity Measure. In *International Conference on Case-Based Reasoning*, Lecture Notes in Computer Science, pages 375–389, June 2017.

[35] Ashish Sureka, Atul Goyal, and Ayushi Rastogi. Using social network analysis for mining collaboration data in a defect tracking system for risk and vulnerability analysis. In *Proceedings of the 4th India Software Engineering Conference*, ISEC '11, pages 195–204, New York, NY, USA, February 2011. Association for Computing Machinery.

[36] Xuan Yang, Daning Hu, and Davison M. Robert. How Microblogging Networks Affect Project Success of Open Source Software Development. In *2013 46th Hawaii International Conference on System Sciences*, pages 3178–3186, January 2013.

[37] Yunwen Ye and K. Kishida. Toward an understanding of the motivation of open source software developers. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 419–429, May 2003.

Appendices

A Randomized repository selection

Table 10: Randomized selection of repositories.

Mining	Name	Created at
OK	ansible/awx	17/5/2017 15:50
mining err	antennapod/antennapod	31/7/2012 10:25
bad ver	apache/lucenenet	27/3/2009 15:41
OK	apinf/platform	4/3/2015 8:33
OK	aria2/aria2	27/11/2010 9:41
bad ver	arvidn/libtorrent	3/6/2015 5:22
OK	astropy/halotools	9/1/2015 21:43
mining err	augurproject/augur-ui	2/2/2015 20:56
bad ver	azure/azure-sdk-for-net	6/12/2011 23:32
OK	betaflight/betaflight-configurator	8/6/2016 22:32
OK	bitzenycoredevelopers/bitzeny	24/10/2017 14:48
OK	boost-ext/di	23/1/2012 21:42
OK	canjs/canjs	20/1/2012 17:57
bad ver	cegui/cegui	9/1/2020 8:15
bad ver	chakra-ui/chakra-ui	17/8/2019 14:27
bad ver	clasp-developers/clasp	20/5/2014 3:54
mining err	corretto/corretto-11	11/2/2019 20:13
OK	dbeaver/dbeaver	21/10/2015 8:26
OK	dgcdev/digitalcoin	8/9/2014 9:58
OK	dhis2/dhis2-android-capture-app	11/4/2018 17:1
OK	digitalgreenorg/dg	9/11/2011 16:4
bad ver	dimagi/commcare-hq	9/7/2009 17:0
OK	dita-ot/dita-ot	14/4/2012 4:49
OK	ember-cli/ember-cli	3/11/2013 17:34
OK	endless-sky/endless-sky	14/3/2015 20:44
mining err	euroelessar/qutim	18/5/2011 14:56
OK	fioprotocol/fio	3/3/2020 20:2
bad ver	firemodels/smv	31/8/2016 15:46
OK	flowminder/flowkit	31/10/2018 23:59
bad ver	foam-framework/foam	1/5/2014 18:10
mining err	frappe/erpnext	8/6/2011 8:20
OK	geli-lms/geli	14/3/2017 13:56

Continued on next page

Table 10 – *Continued from previous page*

Mining	Name	Created at
bad ver	gemini-hlsw/ocs	12/9/2014 14:20
OK	geonetwork/core-geonetwork	14/6/2012 9:59
OK	globalarrays/ga	17/2/2017 20:48
OK	gmod/jbrowse	16/1/2009 11:30
bad ver	gulden/gulden-official	13/8/2015 10:46
mining err	handsontable/handsontable	23/5/2011 22:38
OK	ibm/carbon-components-angular	31/7/2018 19:41
OK	imageengine/cortex	12/6/2013 23:12
bad ver	intermine/intermine	2/7/2012 10:25
bad ver	iotaledger/trinity-wallet	23/5/2018 15:8
mining err	iov-one/iov-core	17/4/2018 18:46
OK	jhipster/generator-jhipster	21/10/2013 20:7
OK	kiwix/kiwix-android	18/1/2017 19:48
OK	ksp-ro/realismoverhaul	5/1/2014 23:2
OK	lavaio/lava	3/6/2019 2:20
OK	ledgerhq/ledger-live-desktop	21/2/2017 12:52
OK	linq2db/linq2db	18/6/2011 16:24
OK	linuxdeepin/dde-control-center	6/11/2013 7:23
mining err	liqd/adhocracy3	15/8/2014 8:7
mining err	liveblog/liveblog	18/2/2015 9:22
mining err	livelykernel/livelykernel	10/2/2012 0:31
bad ver	lk8000/lk8000	9/1/2011 16:55
OK	mavlink/qgroundcontrol	13/10/2011 7:31
OK	megaglest/megaglest-source	25/11/2013 21:10
bad ver	micro-ros/nuttx	15/1/2018 14:29
bad ver	microsoft/azure-pipelines-tasks	6/12/2014 11:11
mining err	microsoft/cntk	26/11/2015 9:52
bad ver	microsoft/recommenders	19/9/2018 10:6
bad ver	mozilla/addons-frontend	19/2/2016 17:25
bad ver	mvvmcross/mvvmcross	28/11/2011 22:45
OK	myhush/hush	16/6/2017 14:37
OK	nccgroup/scoutsuite	30/10/2018 11:46
bad ver	nfs-ganesha/nfs-ganesha	2/11/2012 17:25
bad ver	nightscoutfoundation/xdrip	23/9/2016 13:33
bad ver	nuget/nugetgallery	9/8/2011 17:57
OK	nukeykt/nblood	12/2/2019 13:57
OK	nunit/nunit	18/10/2013 23:43
OK	open62541/open62541	20/12/2013 8:45

Continued on next page

Table 10 – *Continued from previous page*

Mining	Name	Created at
OK	openbazaar/openbazaar-desktop	21/6/2016 14:34
OK	openzfs/zfs	14/12/2009 20:20
mining err	pculture/miro	31/8/2011 15:58
mining err	photonstorm/phaser	12/4/2013 12:27
OK	pmeal/openpnm	25/7/2013 20:32
OK	project-osrm/osrm-backend	22/9/2011 10:5
OK	ptmt/react-native-macos	4/10/2015 15:22
OK	real-logic/aeron	7/2/2014 17:16
OK	reportportal/service-ui	14/9/2016 12:38
OK	restcomm/sip-servlets	17/8/2012 14:7
OK	rptools/maptool	21/10/2014 12:26
OK	scipp/scipp	6/9/2018 7:1
OK	secdev/scapy	1/10/2015 17:6
OK	sefaria/sefaria-project	20/9/2011 20:23
OK	simpeg/simpeg	26/11/2013 19:46
OK	souffle-lang/souffle	12/3/2016 3:39
OK	spongepowered/sponge	11/4/2015 20:38
mining err	spongepowered/spongecommon	11/4/2015 20:38
OK	spotbugs/spotbugs	4/11/2016 22:18
OK	spyder-ide/spyder	16/2/2015 22:49
OK	stack-of-tasks/pinocchio	7/10/2014 15:42
OK	swaywm/sway	5/8/2015 1:31
bad ver	syndesisio/syndesis	2/10/2017 17:26
OK	taskcluster/taskcluster	15/12/2018 3:48
OK	telefonicaid/fiware-orion	13/9/2013 11:55
mining err	theano/theano	10/8/2011 3:48
OK	tropy/tropy	3/11/2015 21:29
OK	tryghost/ghost	4/5/2013 11:9
OK	twbs/bootstrap	29/7/2011 21:19
bad ver	ultimaker/cura	16/6/2014 12:55
OK	unidata/netcdf-c	6/8/2013 20:48
OK	unvanquished/unvanquished	30/9/2011 16:44
bad ver	uportal-project/uportal	20/10/2011 15:34
OK	visit-dav/visit	13/1/2019 23:27
OK	vuetifyjs/vuetify	12/9/2016 0:39
OK	wazuh/wazuh-kibana-app	29/6/2016 1:43
mining err	zeebe-io/zeebe	20/3/2016 3:38
bad ver	zenorogue/hyperrogue	8/8/2015 13:53

Continued on next page

Table 10 – *Continued from previous page*

Mining	**Name**	**Created at**
OK	zephyrproject-rtos/zephyr	26/5/2016 17:54
OK	zilliqa/zilliqa	27/12/2017 7:42