Bachelor's Thesis

NETWORK-BASED CLASSIFICATION OF DEVELOPER ROLES IN OPEN-SOURCE-PROJECTS: AN EMPIRICAL STUDY

NILS ALZNAUER

November 18, 2020

Advisor:
Thomas Bock Chair of Software Engineering

Examiners:
Prof. Dr. Sven Apel Chair of Software Engineering
Prof. Dr. Andreas Zeller Professor for Software Engineering

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

UNIVERSITÄT
DES
SAARLANDES

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,_____ _____

(Datum/Date) (Unterschrift/Signature)

ABSTRACT

For many Open-Source Software (OSS) projects, no official core developers or project leaders can be found. Identifying core developers for these projects is crucial as OSS projects have very high turnovers in their community. To mitigate this turnover and ensure a smooth-running project, consistency in some developers is needed. This consistency is provided by people maintaining the project for an extended period of time. As Open-Source Software is used by many people worldwide due to it being free, there is an obligation for many projects that the program is secure and follows specific quality standards. To identify responsible people maintaining the project, we employ network-based centrality measures on socio-technical networks. To allow for a comparison of all network-based centrality measures, we create in our study a Ground Truth based on issue data from GitHub. This Ground Truth uses the different event types that can be triggered in GitHub by differentiating them into three groups of permission levels and, thus, finding core developers. Comparing the different centrality measures with the Ground Truth shows that different network classification metrics perform best depending on the project size and activity. This comparison further indicates that the combination of both the Issue Networks and Cochange Networks provides the most information for the classification. We notice, however, that different centrality measures lead to vastly different results on the same project. Hence, our findings help in identifying network classification metrics that correctly classify *core developer*s.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

TP	True Positives
FP	False Positives
TN	True Negatives
FN	False Negatives
TPR	True Positive Rate
TNR	True Negative Rate
PPV	Predictive Positive Rate
DTP	Data Transfer Project
GT	Ground Truth
OSS	Open-Source Software
TSC	*Technical Steering Committee*
TLS	Transport Layer Security
SSL	Secure Sockets Layer
OMC	*OpenSSL Management Committee*
OTC	*OpenSSL Technical Committee*
SIG	*Keras Special Interest Group*

1 INTRODUCTION

Software projects and software companies get bigger and bigger. More and more software developers are employed and continue to write code for libraries or applications that can often be used by everybody. In addition to these commercially driven companies, many software developers write code that is protected under specific licenses but is fully accessible to anybody. These are the so-called Open-Source Software (OSS) projects.

OSS projects are a major way to distribute programs to people as they often fill gaps in workflows or provide a free alternative to standing software solutions.[1] Some of the most famous open-source projects include the office suite *LibreOffice*, the browsers *Mozilla Firefox* and *Chromium*, and the *Linux Kernel*. Since this code is freely accessible and changeable by anyone, any developer that takes a liking to the project can work on it. This kind of contribution leads to a very decentralized way of creating code as these developers can be distributed all around the world. To still ensure that the project is a success, much communication must take place. Consequently, many OSS projects use GitHub[2] or similar tools as software development platforms since they enable the project to use issues to track questions and bugs as well as comment on another person's work, create documentation, project boards, etc.

As developers continue to write an ever-growing amount of code, the probability of writing code that will generate bugs or fatal flaws increases. In addition to having bugs in the code that can lead to data leaks and, in the end, to the failure of a project, some of these bugs can even lead to deaths[3,4]. Since bugs can lead to a huge extent of damage[5,6], software companies implement various possibilities [6, 14] to get everything right, from the initial idea for the project to the deployment and following support. To help to prevent software failure, extensive testing of the project and its functions is done. Project leaders are responsible for leading developers through the project and ensuring that the targets are met. Additionally, they set up the working environment for everybody to work together. In OSS projects, there must be people (maintainers) who know what they are doing, that possibly know how to lead a project and that are aware of quality checks and how to work with people. Additionally, it would be best if these maintainers were around and did not

1 For a full definition of open-source, take a look at the official site of the Open-Source Initiative (`https://opensource.org/osd`).

2 `https://github.com/`

3 Crash of American Airlines Flight 965 resulting in the deaths of 151 passengers partially due to an auto-completion error `https://reports.aviation-safety.net/1995/19951220-1_B752_N651AA.pdf`

4 Death of 5 patients due to an overdose during radiation treatment `https://web.stanford.edu/class/cs240/old/sp2014/readings/therac-25.pdf`

5 Approximately 370 Million US $ on the Ariane 5 launch `https://esamultimedia.esa.int/docs/esa-x-1819eng.pdf`

6 Approximately 327 Million US $ on the crash of the Mars Climate Orbiter, due to conversion problems `https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf`

change every other month, so that consistency is kept in the work and code. The difference between developers in OSS projects from all over the world, even if a tool like GitHub is used, is generally still very big and, therefore, leads to many developers leaving the project [9, 10] after some time while other developers will join to help out. Some of these joining collaborators attention may only lie on one single feature they would like to have implemented, while others want to help out with the general work. This focus on specific features or ideas will lead to a high turnover and much evolution of the connection between all developers.

To still keep an OSS project running smoothly, it is thus critical that there is consistency in, at least, some of the developers so that cases like deaths through software bugs are less likely. Furthermore, this consistency is vital to keep people on track and target since only the maintainers know why the targets were set in that way and how they all come together [2]. To achieve all this, the maintainers help others, answer questions, and have a general overview of the project. However, who exactly are these people? Who are the people one can ask if there are misunderstandings? With whom can one discuss where a particular feature should be implemented? Moreover, who will review one's code and decide whether it is good enough to be used?

These questions could be quickly answered in a company since the hierarchy is clear, and everybody knows whom to talk to. For OSS projects, this is not the case. A list of the maintainers and administrators are rarely found, may it be on the website for the project (if one exists) or the GitHub project page itself. From the seven projects discussed in this study, only four had such a list (and those were not visible without making an effort to find them). Moreover, even for these projects, there remains the question of whether this list is up to date or depicts the complete history of former project leaders. To circumvent the problem of not knowing who the leaders are, developers choose measures to help them find the maintainers. Some of them might be:

- Is it the number of commits somebody has made to the project?
- Is it the number of issues this person has worked on?
- Is it the way they answer one's questions?

These are only some ideas of the measures somebody can come up with. However, no one knows whether these measures will lead to a definitive answer and if a measure exists what can be predicted for future evolution?

To understand the evolution of software projects and the role each developer plays, Joblin et al. [12] and Joblin, Apel, and Mauerer [13] classified developers into different categories like *core developer* and *peripheral developer* by using metrics based on counts and networks. Using this distinction into core and peripheral, in this thesis, we compare them to a *Ground Truth* that is created based on GitHub issue data and analyzes how they overlap and, therefore, how useful this distinction is in different OSS projects. To create the *Ground Truth*, we analyze which of the issue events in the GitHub issue data can only be triggered by maintainers and hence filter all issues for these issue events. If there is an official committer list for a project, we also analyze the network classification metric according to this list and compare our *Ground Truth* to the official committer list.

1.1 GOAL OF THIS THESIS

This thesis aims to create a *Ground Truth* for developers that should be classified as core developers based on GitHub issue data depending on the project. Furthermore, we split our data using nine months time windows to retain a dynamic within the networks and our *Ground Truth*. This *Ground Truth* is then used to check the output of different metrics, in our case, count-based and network-based metrics, for their validity and similarity. Additionally, these metrics are evaluated against the actual core developer lists of individual projects, if such a list exists. From this, the following research questions are derived:

> *RQ 1: Which of the network metrics classify core developers correctly when compared with our Ground Truth?*
>
> *RQ 2: Which of the network metrics classify core developers correctly when compared to an official core developer list?*
>
> *RQ 3: Is our Ground Truth comparable to the official core developer lists?*

To answer these questions, we build different networks based on the data of seven different projects. These projects are independently tested, and using network classification metrics, we evaluate each of these projects. For this, we compare the results of the network classification metric against our Ground Truth and the official committer list, respectively. We find that, out of the tested network classification metrics, the eigenvector centrality classification performs best on most projects. Furthermore, we conclude that combining of both GitHub issue data and commit data is the most accurate in classifying core developers.

1.2 OVERVIEW

Starting with a general background in Chapter 2, we give an overview of networks and how they are built. In addition to this, all metrics and their advantages and disadvantages are explained in detail. It is also mentioned how the *Ground Truth* is created. Chapter 4 shows how the metrics and the following evaluation were implemented. Furthermore, all variables that are evaluated are explained. In the following chapter, all projects are described by giving a short overview of their history and some information concerning the list of core developers and possible events that impacted this project. After explaining the projects in Chapter 5, we perform a full evaluation and discussion of the performance of the network classification metric using different similarity metrics under certain variable combinations in Chapter 6. Later, in Chapter 7, there is a discussion of our threats to validity and how this thesis can impact research. Furthermore, we also explain other steps on how future work can be done with the knowledge of the best combination of networks and network classification metrics. Last but not least, we conclude by summing up the most critical takeaways and results.

2

BACKGROUND

In this chapter, we introduce terminology and concepts that are used in our study. We provide general information on these ideas and explain the utilization of each. In Section 2.1, we mention networks, how they are built, and explain their use in a general setting. Additionally, five different metrics are presented, three of which are used to find central nodes in networks. In Section 2.2, the permissions of GitHub are described, as we use these permissions to compute our Ground Truth.

2.1 NETWORKS

In our study, we utilize networks to analyze our data. Networks are used to show connections between objects. These connections can then be analyzed through different algorithms that return a set of objects in each network that satisfy certain constraints.

In a mathematical sense, networks are *graphs* [7, pp. 2–4]. Graphs (G) consist of nodes (V) and edges (E). The node can be any object that the user wants, while the edges are connections mentioned above between these objects. To describe any graph, two sets are needed. One set of nodes and one set of edges:

$$G = (V, E)$$

Every edge is connected to two nodes and is thus described as a pair formed by two nodes taken from the set of nodes:

$$E = \{(v_i, v_j) | v_i, v_j \in V\}$$

In General, there are two different ways a graph can be described. It can be *undirected* or *directed*. The difference being that every edge in a directed graph has a direction from one node to another. Additionally, every graph can be described as a matrix denoting every edge. The rows and columns of the matrix are the nodes, while the values within the matrix describe the number of edges, which is dependent on whether the graph is *directed* or *undirected*. These matrices are presented for each of the two graphs, one being *directed*, the other *undirected*.

First, we take a look at *undirected* graphs. In these graphs, there is no orientation. Having no orientation means that when an edge is introduced between two nodes, there is no information from where this edge starts or where it ends. Therefore, the edges are *unordered* pairs, which means that there is no difference between $(v_1, v_2) \in E$ or $(v_2, v_1) \in E$. Both of them describe the same edge. The matrix A for undirected graphs is represented like this:

$$a_{ij} = |(v_i, v_j)| + |(v_j, v_i)|$$

Directed graphs, on the other hand, introduce a direction and, therefore, orientation for edges. Hence, every edge now has a start and an end node to which they connect. This means that the start point has a connection to the endpoint but not the other way around except for the case that such an edge exists. The edges are thus *ordered* pairs. It is now essential which node is put in the front. The edges $(v_1, v_2) \in E$ and $(v_2, v_1) \in E$ are therefore not describing the same edge. This means that matrix A is built as follows:

$$a_{ij} = |(v_i, v_j)| \tag{2.1}$$

From this point on, we use the term *network* as a synonym for *graph*.

2.1.1 *Socio-technical Network*

In this study, we also use a special kind of network, the so-called *socio-technical network*. This network consists of two datasets where one is based on technical data, while the other uses social data.

In our case, developers and different connections between them make up the network. We consider the developers from the different projects to be the nodes of the network, while the edges are special ways of interaction between these developers. Later on, we differentiate between different networks that are analyzed. Some of them are more in the technical, others more in the social branch of networks. All our networks are explained in Section 4.3.

2.1.2 *Input Data*

To build such socio-technical networks, we use three data sources:

- **Commit data** contain everything about all commits done on a project. This includes the author, e-mail address, and the respective commit hash to make the commits identifiable. This data is used in a technical way, as they only contain information on technical connections between developers.
- **Mailing list data** contain the header information of every e-mail sent to the mailing list. The *mailing list data* contain the author, date, e-mail address, and the subject and the thread to which the e-mail belongs. This data only contains information on social connections between developers.
- **GitHub issue data** contain all issues and pull requests. This means that any comment and any interaction between developers that happened on one issue is taken into account. Our data include the issue number, the title, the author, the e-mail address as well as the current state of the issue or pull request. Additionally, it also contains the date on which each of the issue events (Section A.1) happened as well as the exact event that happened, such as e. g., *commented* or *reviewed*. These data are both technical and social information, as an issue is an easily comprehensible connection of a technical discussion.

Based on the data provided, we are able to create developer networks in many different ways, e. g., using the developer relations *cochange*, *issue*, and *mail*. In addition, we also build developer networks based on combinations of all three relations.

2.1.3 *Count-based Metrics*

Count-based metrics are some of the easiest to compute metrics. In most cases, they can be computed quickly and give a concise summary of the participation a developer had in a project. This, however, does not mean that this participation is meaningful to the project. This leads to great threats to validity because any developer "cleaning" the code, meaning changing the code style, correcting spelling mistakes, and simply moving files or code around, artificially increases their count-based score without adding anything meaningful to the project's organizational workflow.

In the following, we present an overview of the two count-based metrics, *Commit Count* and *Lines of Code Count*. They are explained in a concise manner, including their advantages, disadvantages, and how they are computed.

2.1.3.1 *Commit Count*

Commit Count is a simple metric which, as its name suggests, counts the number of commits for each author. It is not important how much was committed during one of these commits, how many lines were added or deleted, or whether some files were moved to another place in the repository. Every commit is as valuable as any other commit, and no weights are used.

An advantage of this metric is that it is easily computed and understood. However, a clear disadvantage is the fact that all commits, no matter their size or importance for the project, are considered the same. This leads to a problem in which the moving of files or changing the style of code is counted the same as fixing a crucial security bug. Changing code style could, therefore, lead to an artificial increase in the centrality of a developer, putting them at the same level of contribution as someone that contributed a crucial feature to the project.

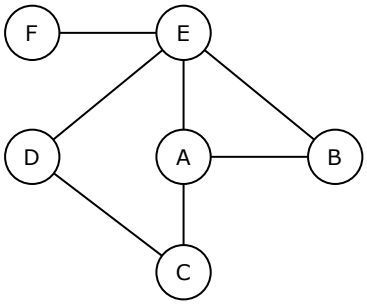
2.1.3.2 *Lines of Code Count*

Similar to *Commit Count*, *Lines of Code Count* is a metric that counts all lines that were added and deleted by an author in all their commits and adds them all up.

As with *Commit Count*, the computation of *Lines of Code Count* is rather easy since we need to iterate over all commits and save the total number of deleted and created lines for each author. This metric can be easily understood but has the same glaring disadvantage as *Commit Count*.

2.1.4 *Network Classification Metrics*

Network classification metrics are very different from *Count-based metrics* in a way that they are not computed as easily in most cases. They provide a more in-depth understanding of



Node (v)	$deg(v)$
A	3
B	2
C	2
D	2
E	4
F	1

Figure 2.1: An example network for the values of Node Degree using an undirected network.

the underlying data and the different characteristics of the nodes and their connections that were not obvious or even visible before.

We focus mainly on these network metrics since the flaws in the classification of developers through *Count-based metrics* are too relevant, to not consider them.

2.1.4.1 *Node Degree*

One of the simplest metrics for networks is based on the degree of nodes [7, pp. 4–6]. The *degree* of a node is either the total number of outgoing edges, the total number of incoming edges, or simply the number of edges connected to this node. In our study, we use the sum of the outgoing as well as the incoming edges as its degree. For an undirected network, this computation is ignored, and we are given the number of connections of the node. The formula for undirected graphs is as follows:

$$deg(v_i) = \sum_j a_{ij}$$

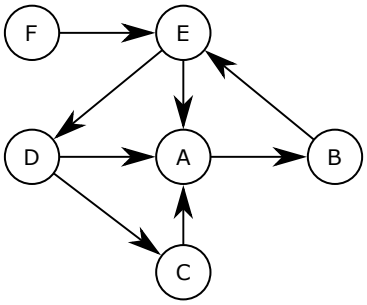
For directed graphs, it is a little different, since to add all outgoing and incoming edges:

$$deg(v_i^{in}) = \sum_j a_{ji}$$
$$deg(v_i^{out}) = \sum_j a_{ij}$$
$$deg(v_i) = deg(v_i^{in}) + deg(v_i^{out})$$

The degree of a node is very useful when it is important to find out which nodes have the most connections in a local setting and, therefore, possibly people that are very interconnected with other people or are doing a lot, depending on the built network.

In Figure 2.1, we present an example of an undirected network. Right beside it is a table that denotes the degree for each of the nodes. As there are only undirected edges, every node simply counts its connections.

In Figure 2.2, we present an example of a directed network in which all outgoing nodes are counted.



Node (v)	$deg(v^{out})$
A	1
B	1
C	1
D	2
E	2
F	1

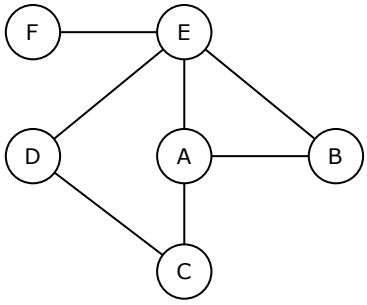
Figure 2.2: An example network for the values of Node Degree using a directed network.

2.1.4.2 *Eigenvector Centrality*

The *eigenvector centrality* [20, pp. 169–172] is a centrality measure that is not as intuitive and easy to compute as the node degree metric. Instead of concerning itself with a "local" setting, eigenvector centrality works on a global level. This is done using the idea that a central node is mostly connected to other nodes that are considered central as well. To get to know the eigenvector centrality, the adjacency matrix is multiplied with a special eigenvector of the adjacency matrix (A). This eigenvector (x) is chosen to be the one that only consists of positive values. To get the eigenvector centrality of a network, this eigenvector (x) is repeatedly multiplied with the adjacency and subsequently normalized. Once this eigenvector diverges, it displays a value for every node within the network and assigns it a centrality value. The higher this centrality value is, the higher the centrality in the network. The following is a formula for the computation of the eigenvector centrality:

$$x_t = A * x_{t-1}$$

The eigenvector at time t is denoted as x_t. When x_t and x_{t-1} converge, the eigenvector centrality for each node is found. In Figure 2.3, we show a network and the eigenvector centrality values for each node.



Node	Eigenvector Centrality
A	0,498
B	0,408
C	0,323
D	0,341
E	0,563
F	0,217

Figure 2.3: An example network for the values of Eigenvector Centrality using an undirected network. The results were obtained using the eigenvector computation from UCINET [27]

2.1.4.3 *Hierarchy*

Another rather simple network metric is *Network Hierarchy* [24]. This network metric tries to find nodes that are connected to many nodes that are well connected to other nodes but at the same time are not connected to each other. This means that the hierarchy is similar to a mathematical tree where the root has connections to its children, but these children do not have any or very few connections between each other. To compute the hierarchy, we use the degree of a node, as mentioned in Section 2.1.4.1 and the so-called clustering coefficient. The clustering coefficient is used to find clusters in networks, i. e., where there are nodes that are connected to nodes that are connected with each other. The formula of the clustering coefficient is as follows:

$$
\begin{aligned}
cc(v_i) &= \frac{2n_i}{k_i(k_i-1)} \\
k_i &= \#\text{neighbors connected to } v_i \\
n_i &= \#\text{edges between all } k_i
\end{aligned}
$$

The clustering coefficient, therefore, shows how well connected all the neighbors are since all k_i nodes can have a maximum number of $\frac{k_i(k_i-1)}{2}$ edges between them (not counting loops of edges to themselves).

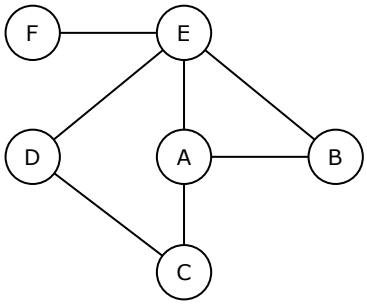
Since we are not interested in the points in the network where people are really clustered but in the people that are highly connected with neighbor nodes that are not interconnected, we divide the degree of a node by its clustering coefficient:

$$
hierarchy(v_i) = \frac{deg(v_i)}{cc(v_i)}
$$

A high centrality value for a network computed with the hierarchy metric is, therefore, created, when a node has a high degree and all its neighbors are not connected. Developers with a high hierarchy value are interesting for us, as these are developers that link small clustered groups together [24]. An example of this would be that there are many small groups within a network denoting issues on which people worked together. The developers with a high hierarchy value are now those people that took part in many of these groups and they are, therefore, the link between them. This coincides very well with the idea of a *core developer*.

An example network and its corresponding hierarchy computation can be seen in Figure 2.4. In the last line of the table, we can see that the value of the clustering coefficient is NaN. The reason for this is, that this node has only one neighbor. The clustering coefficient is therefore not defined and gets the tag "Not A Number". This is propagated to the hierarchy value as well, since a node can not be in a position of hierarchy, when it only has one neighbor. Additionally, in lines 3 and 4 of the table the clustering coefficient is 0. As the hierarchy value would, therefore, be inf and distort our centrality classification. Hence we set every node's hierarchy value to 0 when the clustering coefficient is 0[1].

1 Documentation of CORONET (`https://github.com/se-passau/coronet`)



Node	$deg(v)$	$cc(v)$	Hierarchy
A	3	1	3
B	2	1	2
C	2	0	0
D	2	0	0
E	4	$\frac{1}{6}$	24
F	1	NaN	NaN

Figure 2.4: An example network for the values of Hierarchy and clustering coefficient using an undirected network.

2.2 GITHUB PERMISSIONS

To test whether or not the network classification metrics are sensible, we need to know what should actually be their output. In our study, we create a Ground Truth (see Section 3.1) and test the outputs of the network classification metrics against it. Using different measurements, we evaluate the network classification outputs against our Ground Truth, to suggest a dominating network classification metric.

To compute our Ground Truth, we choose to take a look at the issues and pull requests on GitHub. Issues and pull requests on GitHub have many different meanings and different tasks. Pull requests are used when a developer submits changes to the repository so that other developers can evaluate and review the changes so that the changes can be merged into the official repository. Any developer that does not have *write* permission on a repository has to use this way to get their changes approved and merged into the project. This approval is in most cases done by requesting other developers to review the changes and once the changes are accepted by the reviewers, the pull request is usually pushed to the main branch. Issues, on the other hand help people coordinate and work in the organization. They can also be used to comment on other people's work and serve as a general Q&A board where outside users can reach *core developer*s and discuss new ideas. Issues and pull requests are, therefore, a good way to distinguish between outside users, *core developer*s, and maintainers, since their interaction is very different. Core developers and maintainers for example are able to create pull requests and subsequently merge them onto the main work branch. Outside users on the other hand have to go through a difficult process of getting their pull request approved so that it can then be merged. To make this differentiation, we take a look at the permissions that are implemented on GitHub repositories and the privileges that come with each of these permission levels.

The aforementioned organization in which people can work and discuss, is a shared account that enables the collaboration "across many projects at once" [30]. Additionally, organizations in GitHub have the feature that the owner can hand out permissions on a team level. Furthermore these teams can be structured in the same way a company has structured its hierarchy. This means that a company can be mirrored on GitHub, therefore,

maintaining the outside structure and also the different permission and responsibility levels of different individuals.

However, it is important to know that an organization owner and users with admin permission can limit interactions between users [80]. Furthermore, the organization owner can give certain privileges to users with admin permission. As we do not distinguish between admin users and organization owners in our analysis and as there is no possibility to view all the individual permissions for each project and the individual permission levels for each organization, we assume that all permissions are default and unchanged. The following are the permission levels a person can have on a GitHub repository:

- **Read**. The *read* permission on GitHub repositories is generally "Recommended for non-code contributors who want to view or discuss your project" [86]. This is the default permission for any GitHub user seeing and following a project. In a more concrete matter, users with *read* permission can view the project, open issues, submit reviews on pull requests and communicate with other users through comments and the wiki. Furthermore they are able to fork and send pull requests from their own fork to the main project where it can then be evaluated. Any newly created public repository has the *read* permission as its default permission for any GitHub user.
- **Triage**. The *triage* permission on GitHub repositories is generally "Recommended for contributors who need to proactively manage issues and pull requests without write access" [86]. This permission level can only be given to a user by the owner of an organization for any repository of this organization or an administrator of a repository to this repository only. A user with *triage* permission can do anything users with *read* permission can do and additionally has the ability to apply labels and clean up issues by marking them as duplicate or closing and reopening issues they themselves did not author, as well as assigning issues to other users. They are able to request pull requests from people, meaning assigning them to the task.
- **Write**. The *write* permission on GitHub repositories is generally "Recommended for contributors who actively push to your project" [86]. This permission level can only be given to a user by the owner of an organization for any repository of this organization or an administrator of a repository to this repository only. A user with *write* permission can do anything users with *triage* permission can do as well as push to the repository they are a part of. Furthermore, the user can publish packages and submit relevant reviews on pull requests that "affect a pull request's mergeability" [86]. There are numerous other privileges that do not come into play for issues such as viewing draft releases and are, therefore, not mentioned here.
- **Maintain**. The *maintain* permission on GitHub repositories is generally "Recommended for project managers who need to manage the repository without access to sensitive or destructive actions" [86]. This permission level can only be given to a user by the owner of an organization for any repository of this organization or an administrator of a repository to this repository only. A user with *maintain* permission can do anything users with *write* permission can do and additionally have the ability to push to protected branches. They can "limit interactions in a repository" [86] meaning they can restrict other users from opening issues, commenting and creating

pull requests [88]. Additionally, users of *maintain* permission are able to "configure pull request merges" and "manage topics" [86]. "Configure pull request merges" does not mean that the maintainer can merge a pull request regardless of the number of reviews, it just means that they can change the way this merge is done, e. g., deciding between rebasing, squashing and simply merging the commits [82].

- **Admin**. The *admin* permission on GitHub repositories is generally "Recommended for people who need full access to the project, including sensitive and destructive actions like managing security or deleting a repository" [86]. This permission level can only be given to a user by the owner of an organization for any repository of this organization or an administrator of a repository to this repository only. A user with *admin* permission can do anything users with *maintain* permission can do and manage anything in the project. As they are the administrators they can do anything to the project and have only marginally fewer privileges than the organization owners.

It is very important to note that these permissions are built on top of each other with *read* permission being at the bottom and *admin* permission being at the top. Using this analogy, every developer of a higher level has all the permissions below them as well, i. e., even though it is not specifically mentioned, any developer having admin permission can also comment on any issue like developers with *read* permission can.

3

APPROACH

In this chapter, we introduce how our Ground Truth is constructed and on what it is based. We further introduce the way in which we measure the similarity between our Ground Truth and the outputs from the network classification metrics.

3.1 GROUND TRUTH

The Ground Truth we compute is an important part of understanding the network classification metrics. The idea for this Ground Truth (GT) is to compare it and the outputs of the network classification metrics using the measurements described in Section 3.2.

To compute the GT, we use the GitHub *issue data* (2.1.2) as our input data in conjunction with the GitHub permissions presented in Section 2.2.

The goal of the GT is to differentiate developers on GitHub into two distinct groups, *core developer* (4.2.1) and *peripheral developer* (4.2.2). We define a *core developer*, in this context, as a developer that has contributed to the project and was therefore granted *write*, *maintain* or *admin* permission on the analyzed project. All developers that do not fulfill this criteria are considered *peripheral developer*s. To find all developers in a project that are considered as *core developer*s, we analyze all the issue events that can be triggered by GitHub using issues or pull requests.

Every issue event that is triggered by GitHub has a minimum permission level that a developer needs to have, e. g., only developers with *admin* permission or organization owners can trigger the issue event *user_blocked* by blocking a user from working on a repository. On the other hand, all users that have the *read* permission can trigger the event *commented*, by commenting on any issue within the project, this includes basically every person with a GitHub account. In order to find all the issue events, we analyze the documentation of GitHub and the permission levels provided. A detailed explanation and discussion for each issue event can be found in Appendix A.1.

Since there are multiple issue events that we are not able to place exactly into either the *core developer* group or the *peripheral developer* group, we distinguish all event issues into three different groups.

COMMON PRIVILEGES In *common privileges*, we group all issue events of which we are sure that they can be used by all permission levels, i. e., they are accessible to developers with *read* permission. Since any user with a GitHub account has this permission level, these issue events are not considered for the computation of the GT. An example of such an issue event is *commented*. This issue event is triggered every time someone comments on any issue or pull request.

COLLABORATIVE PRIVILEGES In *collaborative privileges*, we group all issue events of which we are unsure whether they belong to *common privileges* or *elite privileges*. Furthermore, we add *triage* permission level developers to this list as well, as they do not have *write* permission to the project, meaning they do not have the privileges a *core developer* should have. However they do have some privileges, so they can not be put into the *common privileges* category. An example of such an issue event is *assigned*. This issue event is triggered when a developer is assigned to an issue or a pull request. This suggests that the one being assigned is a *core developer* as they are trusted with answering or reviewing this issue. However the one assigning, namely the author, can also only have the rights to help out with the organization within the project.

ELITE PRIVILEGES In *elite privileges*, we group all issue events of which we are sure that they can only be used by permission levels *write* and above, i. e., they are only accessible by *core developer*. All issue events that are grouped in this category are used to compute the GT. An example of an issue event that is in *elite privileges* is *pinned*. This issue event is triggered when a developer having the *write* permission level pins an issue to the top of the issue board. Only three such issues can exist. In Table 3.1, we present all issue events and differentiate them into the three groups of privileges. Using these three groups of privileges, the GT can be computed. Since our GT needs to consist of a list of *core developer*s, we can not utilize the *elite privileges* directly. We therefore analyze the GitHub *issue data*, by getting all events that are found in the *elite privileges* group and put all the authors of these events in a set. This set is then used as the official GT.

3.2 MEASUREMENT

To get an accurate representation of how reasonably the network classification metrics classify developers into *core developer*s and *peripheral developer*s, we are comparing these output lists against our issue-based Ground Truth and the official committer lists that were found for some projects. Simply comparing them will however not yield any meaningful results due to differing sizes of the groups of developers. Overall, we use six different measures that give us meaningful feedback. These different measures are divided into two groups:

The first group consists of the similarity measures *Jaccard Index* and the *Overlap Metric*. They are concerned with a rough overview of the data, using the data and comparing it as a whole. However, they only work on two sets of data.

The second group consists of the classification measures Recall, Specificity, Precision and the *F1 metric*. For this group the data is further processed into False Positives, True Positives, False Negatives and True Negatives.

3.2.1 *Similarity Measure Data Setup*

The data setup for the similarity measures is simple, since two sets are taken into account. The first set consists of the *core developer* list created by the network classification metrics. The

Table 3.1: Categorized Issue Events. A detailed explanation of all issue events can be found in the Appendix A.1

Elite Privileges	Collaborative Privileges	Common Privileges
added-to-project	assigned	automatic-base-change-failed
converted-note-to-issue	demilestoned	automatic-base-change-succeeded
deployed	labeled	base-ref-changed
deployment-environment-changed	marked-as-duplicate	closed
locked	milestoned	comment-deleted
merged	review-requested	commented
moved-columns-in-project	unassigned	committed
pinned	unlabeled	connected
removed-from-project	unmarked-as-duplicate	convert-to-draft
review-dismissed		created
review-request-removed		cross-referenced
transferred		disconnected
unlocked		head-ref-deleted
unpinned		head-ref-restored
user-blocked		mentioned
		ready-for-review
		referenced
		referenced-by
		renamed
		reopened
		review-request-removed
		reviewed
		subscribed
		unsubscribed

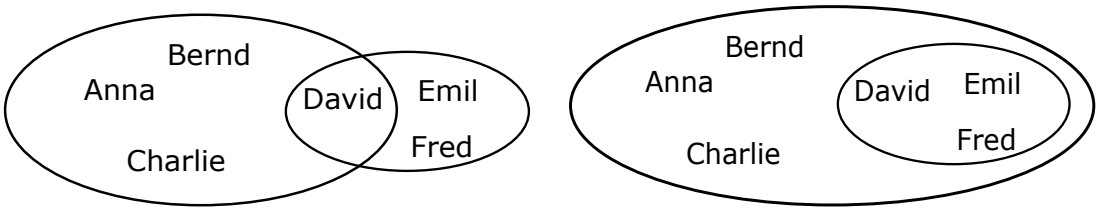
second set consists, depending on the research question at hand, of either the issue-based GT or the official committer list.

3.2.2 *Jaccard Index*

The *Jaccard Index* or *Jaccard similarity coefficient* is a similarity measure named after french botany professor Paul Jaccard. He used this coefficient as a way to describe how similar the distribution of plants is in different areas that have mostly the same conditions while some conditions are vastly different [11]. Today, this similarity measure is not only used in botany but also in computer science and many other scientific fields to show how similar two finite sets are. The formula of the *Jaccard Index* is as follows:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

with A and B being finite sets of data. Should both sets be empty, we set the value to $J(A,B) = 1$. This index can show us very easily how similar our data is while still accounting for different sizes of A and B.



(a) Two sets of names with a small overlap.

(b) A border case in which a big overlap between the two sets appears.

Figure 3.1: An example of two sets of names overlapping.

The following example is based on Figure 3.1a:

$$J(A,B) = \frac{|David|}{|Anna, Bernd, Charlie, David, Emil, Fred|} = \frac{1}{6}$$

In Figure 3.1b, we can see a border case of the *Jaccard Index* in which one of the sets is contained within the other:

$$J(A,B) = \frac{|David, Emil, Fred|}{|Anna, Bernd, Charlie, David, Emil, Fred|} = 0,5$$

Even though one set is contained within the other we still only get a value of $0,5$. This means we lose the information that they are still very similar. In order not to lose this information, we use the *Overlap Metric* which accounts for the differing sizes in the sets.

3.2.3 *Overlap Metric*

The *Overlap Metric* computes the similarity between two finite data sets in a slightly different way as the *Jaccard Index*, since we want to be able to discount the differing sizes [25]. To achieve this, we use the following formula:

$$O(A, B) = \frac{|A \cap B|}{min(|A|, |B|)}$$

with A and B being finite sets of data again. Should one or both sets be empty, the value is set to $O(A, B) = 0$. We use this metric in conjunction with the *Jaccard Index* since we are not able to discern whether we created a GT that included everything returned by the network classification metric or whether the measure was simply bad. As we want to find out whether the output of the network classification metric is a list of *core developer*s, we need to take a look at which of the sets is bigger and if the network classification metrics output is encased in the GT. If this is not the case, we know that the network classification metrics are not doing a good job and we have to find the reason for it.

Computing Figure 3.1a with the *Overlap Metric* results in:

$$O(A, B) = \frac{|David|}{min(|Anna, Bernd, Charlie, David|, |David, Emil, Fred|)} = \frac{1}{3}$$

We find that for the border case (Figure 3.1b), the difference to the *Jaccard Metric* is even clearer:

$$O(A, B) = \frac{|David, Emil, Fred|}{min(|Anna, Bernd, Charlie, David|, |David, Emil, Fred|)} = 1$$

We can see that through the usage of both the *Jaccard Index* and the *Overlap Metric*, we get vastly different information on the same data and the actual similarity between the two sets.

3.2.4 *Classification Measures Data Setup*

The data setup for the classification measures [23] consists of different parts. In general, a given data set is divided into two distinct sets, one containing the data that is considered relevant and one containing the data that is considered irrelevant for a certain data classification. In most cases, a classification results in a subset of the original data consisting of relevant and irrelevant data points. The ratios between correctly classified elements and incorrectly classified elements can then be compared between different classification tasks as they additionally concern themselves with the size of the data sets. Every data point that is in the classification result and is within the relevant data points (i. e., all the data points that were correctly classified as relevant) is called True Positives (TP). All data points that are in the classification result and are in the irrelevant data (i. e., all data points which were incorrectly classified as relevant) are called False Positives (FP). All remaining data points in the relevant set (i. e., the ones that should be classified as relevant, but are not) are called False Negatives (FN) and all the remaining data points in the irrelevant data set (i. e., the ones that are correctly classified as irrelevant) are called True Negatives (TN).

In our setup, the interpretation of these groups is slightly different. For the comparisons done, we evaluate the distinct sets (core and *peripheral developer*s) against the classification result (GT) and not the other way around. This also means that certain border cases need to be taken care of and that the terms TP, FP, TN and FN have slightly different definitions:

1. **True Positives** The true positives are all the data points that are contained in the *core developer*s set identified by using network classification metric and are also included in the GT, i. e., all the correctly as *core developer* classified developers.
2. **False Positives** The false positives are all the data points that are situated in the *core developer*s set identified by using network classification metric but not in the GT, i. e., all the developers incorrectly classified as *core developer*s.
3. **True Negatives** The true negatives are all the data points that can be found in the *peripheral developer*s set but not in the GT, i. e., all the developers that were correctly classified as *peripheral developer*s.
4. **False Negatives** The false negatives are all the data points that are found in the *peripheral developer*s set and in the GT, i. e., all the developers that were incorrectly classified as *peripheral developer*s.
5. **Border Case** The problem that can arise in our data setup is the fact that the GT can contain developers, which are not accounted for in either the *core developer* or the *peripheral developer* list. This leads to incorrectly computed measures. The reason for this problem is because the GT only works on the GitHub *issue data*. This does not pose a problem when using GitHub *issue data* however every case in which the GitHub *issue data* is not used as an input, e. g., in a Mail Network, the computation can potentially result in failure. To circumvent this case, all data points from the GT that are not found in either the *core developer*s nor the *peripheral developer*s, are added to the *peripheral developer*s. They are therefore classified in the FN category.

3.2.5 *Recall (TPR)*

The classification measure Recall (also called True Positive Rate (TPR)) computes the ratio of the correctly as relevant identified data points to the number of all as relevant identified data points. The formula is as follows:

$$TPR = \frac{TP}{TP + FN}$$

In the case of this study, Recall computes the ratio between the developers that are correctly identified as *core developer*s and all the developers that are part of the GT. This is a very important measure, as it shows whether the network metric identifies more relevant or irrelevant data points as relevant. This means that for us a high Recall value is desirable as this shows, that most of the *core developer*s that are in the GT are also found by the network classification metrics.

Using the classification in Figure 3.2, we compute a Recall value of:

$$TPR = \frac{|Charlie, David, Emil|}{|Charlie, David, Emil| + |Fred|} = 0,75$$

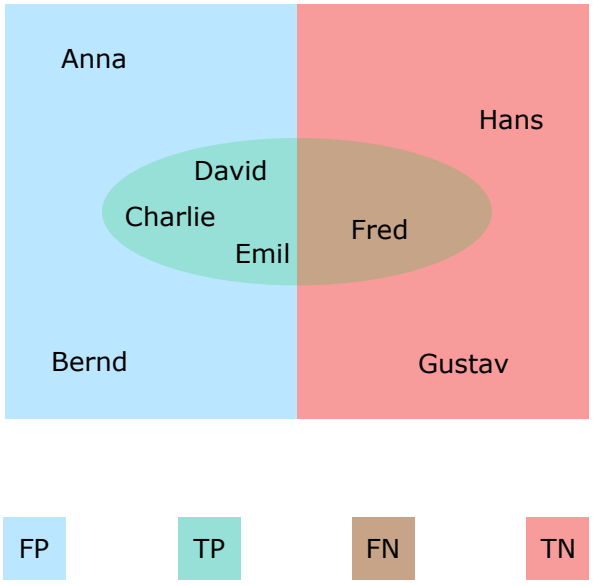


Figure 3.2: An example of a group of *core developers* consisting of True Positives and False Positives, and a group of *peripheral developers* consisting of True Negatives and False Negatives. The computed Ground Truth consists of True Positives and False Negatives.

3.2.6 *Specificity (TNR)*

The classification measure Specificity (also called True Negative Rate (TNR)) computes the ratio of correctly as irrelevant identified data points to the number of all as irrelevant identified data points. The formula is as follows:

$$TNR = \frac{TN}{TN + FP}$$

In the case of this study, Specificity computes the ratio between the developers that are correctly identified as *peripheral developers* and all those developers that are not part of the GT (this includes both *core developers* and *peripheral developers*). This is another important measure, as this shows whether the network metric identifies more relevant or irrelevant data points as irrelevant.

This means that for us a high Specificity value is desirable as it shows that most *peripheral developers* are correctly identified as such.

Using the classification in Figure 3.2, we compute a Specificity value of:

$$TNR = \frac{|Gustav, Hans|}{|Gustav, Hans| + |Anna, Bernd|} = 0,5$$

Overall it is best, if Specificity as well as Recall have both very high values as this shows that the classification works really well.

3.2.7 *Precision (PPV)*

The classification measure Precision (also called Predictive Positive Rate (PPV)) computes the ratio of correctly as relevant identified data points to the number of all that should be identified as relevant. The formula is as follows:

$$PPV = \frac{TP}{TP + FP}$$

In the case of this study, Precision computes the ratio between the developers that are correctly identified as *core developer*s and all those that are classified as *core developer*s by the network classification metric. This measure shows how many of the ones we identified as *core developer*s are actually identified as such.

We therefore hope for a high Precision value as it shows that out of those that are identified as *core developer*s most actually are.

Using the classification in Figure 3.2, we compute a Precision value of:

$$PPV = \frac{|Charlie, David, Emil|}{|Charlie, David, Emil| + |Anna, Bernd|} = 0,6$$

3.2.8 *F1 score (F1)*

The classification measure *F1 score* computes the harmonic mean of TPR and PPV. The formula is as follows:

$$F_1 = 2 * \frac{PPV * TPR}{PPV + TPR}$$

The *F1 score* is another important measure for our study as it mainly takes the values of relevant data into account as opposed to looking at relevant and irrelevant data at the same time. As it could be the case that the irrelevant data is huge compared to the relevant data, measures on both of them can have misleading results. A high value in the F1 score therefore means that both TPR and PPV are high, suggesting that the network classification metric is good at finding relevant data, even though the irrelevant data might be way bigger.

Using the classification in Figure 3.2, we compute the *F1 score* with the help of TPR and PPV:

$$F_1 = 2 * \frac{0,6 * 0,75}{0,6 + 0,75} = \frac{2}{3}$$

4

IMPLEMENTATION

In this chapter, we explain how all our results are computed. For this we first provide a workflow and introduce Coronet, after which we show how the networks are build and which variables are chosen in which way.

4.1 COMPLETE SETUP

The complete setup of our workflow consists of three main parts: (1) the collection of all the data, (2) processing the data and building networks using the tool Coronet, and (3) the analysis of the resulting data.

4.1.1 *Data Collection*

The data collection is done by two tools, namely codeface[1] and codeface-extraction[2]. codeface is a tool that can analyze and extract data from version control systems like GitHub and from *mailing lists*. codeface-extraction is an extension to codeface with the possibility to extract data from the database built by codeface.

The *commit data* is extracted from the official GitHub page of the project using codeface, codeface-extraction and the GitHub REST API[3]. This data includes all commits that were ever done on this repository. The data contains the time of the commit, the author, their e-mail address and the commit hash. More data can be extracted using GitHub but this data is not relevant for building the networks so it is discarded.

The GitHub *issue data* is extracted from the official GitHub page of the project using codeface, codeface-extraction and the GitHub REST API. This data includes all issues and all pull requests as well as anything that was a contribution to the issue, e. g., comments and other interactions with the issue. The GitHub *issue data* contain the issue number, the title, the author, their e-mail address as well as the current state of the or pull request. It further incorporates the issue event type as well as the date of each event. An event can for example be "commented" or "reviewed".

The *mailing list data* are extracted from *gmane*[4] using codeface and codeface-extraction. This data contains the header information of every e-mail sent to the mailing list. This includes the author, the date, the e-mail address as well as the subject and to which thread this e-mail belongs.

1 http://siemens.github.io/codeface/icse2017/#/home
2 https://github.com/se-passau/codeface-extraction
3 https://docs.github.com/en/free-pro-team@latest/rest
4 https://gmane.io/

After the extraction of the data, disambiguation of all author names and their respective e-mail addresses takes place. Since this is sometimes not enough, we also do a manual disambiguation on the data to eliminate any problems.

4.1.2 *Running Coronet*

After the creation of *commit data*, GitHub *issue data*, and *mailing list data*, we use this data with the *R*[5] library Coronet[6] to build different networks based on this data.

4.1.3 *Data Analysis*

The networks created by Coronet are then analyzed and compared against our Ground Truth (3.1) and the official committer lists using the measurements that were mentioned before (3.2).

4.2 CORONET

Coronet is a "R library for configurable and reproducible construction of developer networks"[7]. It is able to build developer networks in a "socio-technical" way, trying to give a good overview over connections between various developers in a team and determining which of these developers is in comparison more central or important for the project's structure. Coronet was developed by the Software Engineering Chair of Professor Apel and is based on the papers of Joblin et al. [12] and Joblin, Apel, and Mauerer [13]. In these papers, they describe the idea of how developers can be classified and suggest networks as a base. For an analysis of the networks, the network metrics Node Degree (2.1.4.1), Eigenvector Centrality (2.1.4.2), and Hierarchy (2.1.4.3) are specifically explained and used to show that the network classification metrics achieve better results in classifying developers than the count-based classification metrics. Coronet implements exactly these ideas for testing and analysis purposes. It has thus many possibilities to create and analyze networks in different ways. However for us, the most important metrics are only network classification metrics as opposed to the network classification metric and the count-based metrics.

Hence, Coronet gives us the possibility to analyze the data and networks so that we can classify all the people in the network into *core developer*s and *peripheral developer*s. To achieve an understanding of the results of Coronet, we need to understand the difference between *core developer*s and *peripheral developer*s.

Nakakoji et al. [19] argue that in an OSS project there are eight different roles a member can be. These eight roles are:

- **Passive User**
- **Reader**

5 https://www.r-project.org/
6 https://github.com/se-passau/coronet
7 Documentation of Coronet: https://github.com/se-passau/coronet (2020-10-15)

- **Bug Reporter**
- **Bug Fixer**
- **Peripheral Developer**
- **Active Developer**
- **Core Member**
- **Project Leader**

These roles are very differentiated, however we are looking at some of these roles or rather at combinations of them. They can be condensed into two main groups, namely our *core developer*s and *peripheral developer*s.

4.2.1 *Core Developers*

*Core developer*s are the people that actively take part in the development of the project and have contributed to it. By default, CORONET sets developers as *core developer*s if they are "in the upper 20th percentile" [13]. This threshold is further explained in Section 4.3.4. To this group we count the *Active Developers*, *Core Members*, and *Project Leaders*. These three groups have in common that they are the most involved people that also have the most rights on GITHUB and the most say in decisions concerning the direction the project should take. They are the most important people as they are the ones that keep the community active and develop most of the code. Additionally to the aforementioned three groups, we also count *Maintainers* as *core developer*s. There is a slight difference between *Core Members* and *Maintainers*. *Maintainers* in general are the developers that are tasked with maintaining the project. This means that they are responsible for "maintaining the infrastructure of the project and support and coordinate all development activities" [8]. As this also coincides with the idea of a *core developer*, we count the *Maintainers* to them as well. In general, we can say that the *core developer*s are the most important developers in an OSS project, as without them, there is a 59% chance that the project fails and development possibly stops [1].

4.2.2 *Peripheral Developers*

*Peripheral developer*s are the developers that sporadically work on the project and may help out with simple code fixes or occasionally adding new functionality and features to the project. The most important word here is "occasionally". If *peripheral developer*s were people that constantly worked on the project they would have to be described as *core developer*s. We will also count *Bug Reporters* and *Bug Fixers* as part of the *peripheral developer*s, since they take part in the project, even if it is only a small amount.

4.2.3 *Other Developers*

As for the last remaining groups *Passive User* and *Reader*, they will be discarded since they are not in the scope of any of our *Research Questions*. This is due to the fact, that they do not

interact with the project in a way that is measurable (using our data), and that they do not have any influence on the direction of the project.

4.3 BUILDING THE NETWORKS

All our networks are built based upon the data collected by CODEFACE. CORONET uses the *igraph* library for building the networks. To build the networks, both nodes and edges are needed. The node set is set up by getting all authors depending on whether *cochange*, *issue*, *mail* or a combination of the three is taken as the network relation. The edges are similarly set up as they take into account the network relation. Any connection between two authors is taken and created as an edge. These connections are explained in more detail in the corresponding network section (*Cochange* (4.3.5.1), *Issue* (4.3.5.2), and Mail (4.3.5.3)). Duplicate edges remain as we are interested in all connections between authors. Following the creation of all nodes and edges, the network is built based on the nodes and edges. In the following section, we explain additional constraints that can be put in place to build networks and analyze them.

4.3.1 *Directed vs. Undirected*

As explained in Section 2.1, any network can be built with two different edge types, *directed* or *undirected*. In our case, the connection between two authors is made if they have worked on the same issue, commit, or e-mail. An undirected edge is created between all authors that worked on the same issue, discussed on the same e-mail thread or worked on the same code artifact. For directed this is slightly different: When author B worked on the instance after author A then there is a connection from author B to author A but not from author A to author B. Should author A for example answer to the comment written by author B at a later stage, then there will be an additional edge from author A to author B.

For our networks, we choose *undirected* as our default setting, as there is not a big difference between either, when looking at the evaluation of the network classification metrics.

4.3.2 *Simplified vs. Unsimplified*

After having built a network, it can be simplified. Having a simplified network depends on two criteria:

1. No loops are contained in the network. A loop is defined as an edge that connects a node to itself. Using the designation from Section 2.1, a loop is defined as

   $$(v_1, v_1) \in E$$

   In a developer network, a loop means that a developer has answered again to an instance of an e-mail thread, issues or code artifact that this developer has already worked on or commented on.

2. There exists only one edge between two different nodes.

When creating networks in CORONET with parameter `simplify` set to `TRUE`, both loops and multiple edges are removed by deleting the edges which are duplicate except for one and removing any edges that loop.

For our networks, we choose *unsimplified* networks as simplifying the networks removes edges without adding weights to the remaining one, meaning we lose connections between people and therefore information on the number of times they worked together. An example of why simplifying the network is bad would be the case that everybody on the mailing list answers to the same welcome mail. This would mean that the person writing the answer now has a connection to all people on the mailing list. With this fact, a developer that was very active on the mailing list would have the same number of edges as the newly joined person, which would defeat the purpose of creating the networks.

4.3.3 *Time Window*

The *time window* is applied to the data beforehand, as we have multiple years of data for every project. For some it is even multiple decades. Using this data and building the networks and then computing the count-based (2.1.3) and network (2.1.4) metrics would result in clear examples of people that continuously worked on the project and most likely did a lot for its success. One goal of our study is to showcase a dynamic in the evolution of *core developer*s. This cannot be done for one data block and one single time window as all information of evolution is lost in such a case. Most literature suggests a time window of three months for the analysis of a socio-technical network [16], as this enables a clear evolution while still having a puffer to make accurate statements. For us, this time window is too short. Using a nine-month time window is more promising, as all projects are OSS and not every *core developer* needs to have a high amount of interaction on GitHub. This lack of interaction would lead to us classifying them as *peripheral developer*s while it is possible that at the same time they took some time off from the project or simply concentrated on a special feature that did not include such interaction. However, having a time window with more months leads again to the problem of losing visible evolution in the networks.

4.3.4 *Core Threshold*

In 1909, Vilfredo Pareto stated in an article that an estimated 80% of consequences come from 20% of the causes [22]. This was later pronounced as a principle that not only applies in economics, but also in many other sciences. One of them being computer science. In multiple papers, researchers discovered that this *Pareto principle* is found in many ways. Some of them are, that 20% of the developers in a project are responsible for 80% of the code or that the writing of 20% code takes up 80% percent of the overall development time and that 80% of all bugs in a project come from 20% of the code [5, 18]. These are only some of the occurrences of the *Pareto principle*, however for us the first case of having 20% of all developers developing 80% of the code is very interesting.

We therefore use this exact principle to identify our *core developer*s. This is done by creating a list of all developers and sorting them by the value they were assigned by the network classification metric. After this, we iterate over the list and add up the value that was

computed for them using the network metrics. When the sum of these values surpasses 80% of all work done, the list of all the remaining developers is classified as *peripheral developer*s, while all others are classified as *core developer*s.

4.3.5 *Cochange vs. Issues vs. Mail*

The three different relations represent connections and interactions of and inbetween people on the project. However they are created on vastly different data. There is, for example, no inherent connection between commits and mails. There might be one such as a mailing thread concerning itself with a bug fix which is therefore connected to the commit containing this bug fix. But this is not guaranteed, as the mailing list can also be used for discussing something that does not result in code changes. It is important to look at all the data sources and not only one of these, since a lot of information can be lost when a large chunk of connections is not taken into consideration. An example is, that a large number of developers was crucial in developing and discussing changes to the project, however only one of these developers was tasked with implementing it. The information of who helped develop the changes would therefore be lost. To counteract the possible loss of information, we create all the networks and analyze their computed network classification metric on how similar they are to the Ground Truth. For this, we will not only look at the three main networks, but also at any combination of the three. This is done to prevent the already mentioned loss of data and connections between developers. An advantage of this approach is that all information is in one network and that this network can then be analyzed and the developers classified. A disadvantage however could be the fact that due to one of the data sets being extremely large and the following disparity in numbers of edges generated, one data set could dominate the other. Since the networks are not weighted, this could mean that we have data sets whose data is basically irrelevant compared to other data sets.

4.3.5.1 *Cochange*

Author networks, which are built using only *commit data*, are created by taking any code artifact (i. e., file) and associating it with all the developers that work on it.

The authors are the nodes of the network and any connection between authors based on the analyzed code artifacts is created as an edge between these two authors. Such a connection is present, when both authors "change[d] the same source-code artifact"[8]. This means that they, for example, worked on the same file within a certain time frame (i. e., Section 4.3.3).

4.3.5.2 *Issues*

For networks, built using GitHub *issue data*, Coronet will use a similar technique as in *Cochange*. Coronet collects all developers that ever triggered an event on an issue or a pull request. The set of these developers then becomes the set of nodes.

Building the network is done by taking the set of all nodes and creating the edges every time two authors have a connection. Such a connection exists, when two authors contribute

8 See footnote 7.

to the same issue or pull request[9]. This can mean that they commit some changes or even just comment on the idea. Basically, every developer that contributed to an issue or pull request has at least one connection to every other developer within this issue or pull request.

4.3.5.3 *Mail*

The networks built on the *mailing list data* are called Mail Networks. As with the other relations, CORONET groups the e-mails by the discussion. This means that two developers that "contribute to the same mail thread are connected with an edge"[10].

Building the author network results in nodes describing the authors and edges that describe the connections between the different mails within the discussions.

4.3.6 *Combined Networks*

All combinations of the three aforementioned networks are similarly built as the base networks. They are first created individually and then merged. This merging is happening on the sets of nodes and the list of edges by taking the union of all nodes and the union of all edges. After merging all the data, a new network is built using the combined set of nodes and the combined list of edges. In this network, we thus have all the edges that were, for example, created based on source-code artifacts as well all the edges that were due to a contribution to the same issue or pull request.

9 See footnote 7.
10 See footnote 7.

5 PROJECTS

In this chapter, we provide an overview of all projects that are chosen for our study. After giving some general information, we introduce each project and mention if mailing lists or official committer lists exist.

5.1 GENERAL INFORMATION

For our study, we choose seven projects that are based on GitHub. These projects are chosen for different reasons, one of them being, that most of them are very prominent so that there is also a lot of data that can be analyzed and used to build the networks. Additionally, these prominent projects are mostly still under development and have therefore a high amount of social interaction through issues and pull requests. Although these are some of the most prominent OSS projects on GitHub, they have vastly differing sizes, with two having a line count of below 100.000 lines, while the rest surpass 1 million lines of code. By studying projects with different sizes, we can find out, whether the network classification metrics classify the developers correctly on big projects as well as small and intermediate ones. Furthermore, we only choose projects that actually use the GitHub issues extensively and do not outsource this kind of communication to other issue and project tracking software like Jira[1]. Last but not least, we choose these projects because they differ in their scope of application so that we can analyze a broader range of applications of OSS projects.

Nextcloud and ownCloud are a little different from the other projects, as Nextcloud is a fork of ownCloud. We expect that we are able to see this developer dynamic when many developers change in ownCloud.

In Table 5.1, we describe the general information for all projects that are analyzed. The columns "#Authors (Commits)" and "#Authors (Issues)" denote the number of all unique developers that triggered an event on commits or issues and pull requests respectively. The column "First Project Activity" contains the first time any event was triggered on GitHub or other version control systems. Since we use different data sources within our analysis, it is sensible to constrict the data to times when all data sources were available so that an actual comparison between all of them takes place. The last column "Latest Data" denotes the last date that all sources within a project had data. When a data source is completely void of any data, this data is not taken into account for the computation of the last date. For all of our projects, we use the *commit data* and GitHub *issue data* found on the corresponding project page on GitHub. For OpenSSL and ownCloud, we additionally use the *mailing list data* since these are the only two projects for which such a mailing list exists.

1 https://www.atlassian.com/software/jira

2 This statistic was generated using the GitStats tool: http://gitstats.sourceforge.net/

Table 5.1: General statistics for every project.[2]

	#Authors (Commits)	#Authors (Issues)	#Commits	#Files	#Lines of Code	First Project Activity	Latest Data
Angular	1.405	22.903	19.270	9.427	1.254.957	2014-09-18	2020-09-25
Data Transfer Project	43	77	1.645	809	68.100	2017-03-20	2020-04-24
Keras	906	13.468	5.350	16	521	2015-03-28	2019-11-06
Nextcloud	890	9.542	55.946	9.525	1.101.667	2010-03-10	2020-09-22
NodeJS	3.183	13.556	32.097	32.046	9.512.093	2009-02-16	2020-02-22
OpenSSL	673	3.497	27.382	23.852	1.254.807	1998-12-21	2019-12-21
ownCloud	614	10.242	44.479	19.355	1.850.217	2010-03-10	2019-10-29

This does not mean that all these projects only communicate via GitHub. However, we are not able to use these communication channels, as this is not implemented within CODEFACE.

5.2 ANGULAR

Angular "is a platform and framework for building single-page client applications using HTML and TypeScript"[3]. Formerly known as *AngularJS*, Angular was redeveloped completely in 2016 when it started in version 2.0.0. It is currently mainly developed by a team at Google while still working under an OSS license which leads to a strong "opinionated" community working on the project [15]. Angular has a very simple system behind it which helps out in different ways. It is a framework that is mainly based on the idea of components that can be reused at other places. To achieve this the framework is built such that all components must be in a parent-child relation, i. e., building a tree with one root component. Additionally, every one of these components must be self-sustaining so that it can be used in other components therefore reducing the amount of code that needs to be written.

3 `https://angular.io/guide/architecture` (2020-11-12)

An official committer list that consists of 30 people was found on their website[4]. They do not provide a definition for their official committer list or the exact position they hold at Angular. Further information and the code can be found on their website[5] and GitHub[6].

5.3 DATA TRANSFER PROJECT (DTP)

The Data Transfer Project (DTP) is "an open-source, service-to-service data portability platform so that all individuals across the web could easily move their data between online service providers whenever they want"[7]. The development goal of the DTP is to enable an easier way for users to transfer data between different online services. The problems with not having a simple way for distributing and exchanging data when wanted are many. Some of the most prominent is that for any such scenario of sending and receiving data between different online services, both services need to have a "separate analysis of the systems, data and human aspects (involving syntactic, semantic and policy facets)"[8]. But not only do these aspects have to be taken care of, but security and privacy concerns need also be tackled to create a safe environment for anybody to use.

To create such an environment and help users with their simple data transport, the DTP creates and builds "adapters" that take in the data from different online services, extract all that is needed, format it into common and widely used data formats and forward this information to another adapter so that this other service can use the data. Using these adapters and a bridge in between to handle the data, the project allows providers to use their current authentication systems as well as not making their own security system open to attacks from the outside [29].

For the DTP no official committer list was found and we can therefore only work with our GT. All the data for the project can be found on their website[9] and on GitHub[10].

5.4 KERAS

Keras is a deep learning API used by many around the world[11]. It is written in *Python* and uses the machine learning platform TensorFlow[12] as its base. Due to an additional focus on high scalability and the ability to use it across platforms, Keras is able to find a liking in research groups and companies that want to use machine learning on large data sets. After its release in 2015 by Francois Challet a Google engineer, it is still completely OSS and usable by anyone with regular updates to the code base. Initially, it was built to be able to use different machine learning platforms but moved to mainly support TensorFlow later. The idea behind Keras is to give its users the ability to quickly comprehend and use

4 https://angular.io/about?group=Angular (2020-10-10)
5 https://angular.io
6 https://github.com/angular
7 https://datatransferproject.dev/
8 See footnote 7.
9 See footnote 7.
10 https://github.com/google/data-transfer-project
11 https://keras.io/why_keras/ (2020-11-04)
12 https://www.tensorflow.org/

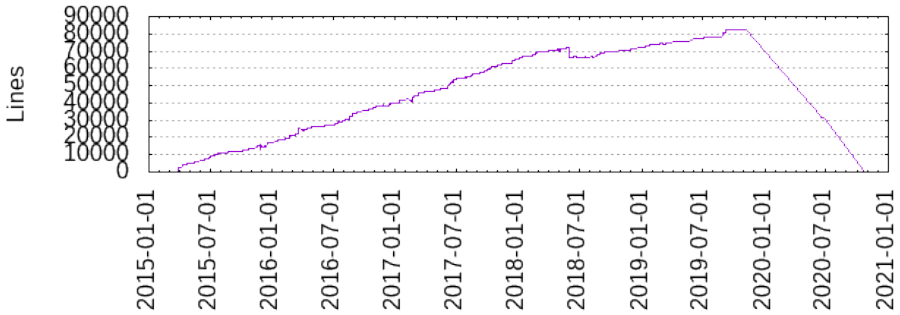


Figure 5.1: The lines of code in project Keras plotted over time. [15]

it. To achieve this, it uses simple APIs for very common use cases and gives relevant and understandable feedback should the user make an error in his handling of data.[13] As a way to make the usage even easier, it is part of a big collection of tools, many of which are related to either Keras or TensorFlow, that help with every possible aspect of using machine learning, may it be data collection or other training models. As of version 2.3.0 which was released on 17th September 2019, Keras no longer provides a multi-backend. This leads to a sharp decrease in files and lines of code as Keras is from that moment on in sync with the *tf.keras* API provided by TensorFlow[14]. This progress can be clearly seen in Figure 5.1.

An official committer list that consists of 5 people was found on their website[16]. These 5 people are part of the *Keras Special Interest Group* (SIG). This *Keras Special Interest Group* is divided into three parts: (1) the BDFL, (2) the Key stakeholders, and (3) Core Keras community contributors. These parties are shortly explained with the result, that the *BDFL* always has the final call in decisions related to Keras, the *Key stakeholders* are responsible for the representation of the project's interest, but not the coding of the project itself and the *Core Keras community contributors* are responsible for running the project.[17] We therefore discard the *Key stakeholders* from the official committer list, as they are not represented within the project, may it be commits or issues and pull requests. Further information and the code can be found on their website[18] and GitHub[19].

5.5 NEXTCLOUD

Nextcloud is an online file syncing and sharing solution directed at enterprises and users able to set up their own servers. It was established in 2016 when the lead designers of ownCloud forked the project because they wanted it to go in another direction. The idea behind Nextcloud is to return control of all the data to the user. This is done by providing

13 See footnote 11.
14 `https://github.com/keras-team/keras/releases/tag/2.3.0` (2020-11-04)
15 See footnote 2.
16 `https://keras.io/governance/` (2020-11-04)
17 See footnote 16.
18 `https://keras.io`
19 `https://github.com/keras-team/keras`

tools so that users can have all their data in their own data centers and on their own servers without having to trust outside sources[20]. Furthermore, Nextcloud helps users by giving them the possibility to integrate already existing cloud services like Google Drive, Dropbox, etc. so that no data needs to be moved around unnecessarily while still having a high degree of security. This is done by having a file access layer that provides the user with the ability to use the storage of all existing cloud services as well as their own personal storage, in one place as well as accessing it from different places and clients [90]. Additionally, to the ability to use most existing infrastructures, Nextcloud has the goal of making its program and storage access as secure as possible. To create such a high-security product, they use the idea of *Two heads are better than one*, which means they made their code complete OSS so that anybody in the world can see and analyze it and report bugs with high Bug Bounty Rewards[21], showing that they are invested in the security of their product. This has so far lead to many developers taking part in the project and creating a project that is secure and free to use.

For the Nextcloud project no official committer list was found, we can therefore only work with our GT. All the data for the project can be found on their website[22] and on GitHub[23].

5.6 NODE.JS

"Node.js is an open-source, cross-platform, JavaScript runtime environment. It executes JavaScript code outside of a browser"[24]. This is the goal of the NodeJS platform and to be able to accomplish such a goal, the developers implemented the ability to run some *JavaScript* code in the back-end. Most platforms use a model in which the website that is run needs to perform computations but at the same time needs to accommodate a high number of simultaneous users. Even if the computations are rather small, they lead to a problem for the server CPUs and therefore a higher latency for the user. NodeJS goes a different way and instead of using threads for every user, it uses a single thread which is non-blocking and event-driven. In general, this is best used when the website needs to have a lot of throughput without having heavy computations that need to be done [28].

An official committer list that consists of 103 people was found on GitHub[25]. This list consists of (1) the *Technical Steering Committee* (TSC), and (2) the collaborators. The *collaborators* are defined as the developers that "maintain the [...] repository"[26]. They are also responsible for reviewing and commiting pull requests to the main branch. The TSC is a subset of the *collaborators* and is mainly responsible for the project governance and the technical direction of the project.[27] We count both of these groups to the group of *core developer*s. Further information and the code can be found on their website[28] and GitHub[29].

20 `https://nextcloud.com/about/` (2020-11-04)
21 See footnote 20.
22 `https://nextcloud.com/`
23 `https://github.com/nextcloud`
24 `https://github.com/nodejs/node`
25 See footnote 24.
26 `https://github.com/nodejs/node/blob/master/GOVERNANCE.md` (2020-11-04)
27 See footnote 26.
28 `https://nodejs.org/`
29 See footnote 24.

5.7 OPENSSL

OpenSSL is a library to enable websites to have secure communications with one another using the Transport Layer Security (TLS) protocol. TLS was formerly called Secure Sockets Layer (SSL) from which the name OpenSSL is derived. As TLS is a widely used protocol for secure network communication, it was established in 1998 as an OSS project. Additionally to creating a secure protocol, OpenSSL also functions as a general-purpose cryptographic library[30].

An OpenSSL that consists of 18 people was found on their website[31]. The OpenSSL project has different groups of management committees, namely (1) the team of committers, (2) the *OpenSSL Management Committee* (OMC), and (3) the *OpenSSL Technical Committee* (OTC). The OMC is defined as "the official voice of the project"[32]. The OTC represents "the official technical voice of the project"[33]. Last but not least the *team of committers* consists of "the people who can commit changes to the OpenSSL source tree" as long as this is done "with appropriate code reviews".[34] As all members of the OTC and the OMC are also part of the *team of committers*, we simply take the *team of committers* as the official committer list. Additionally, we were able to find a mailing list for OpenSSL. Further information and the code can be found on their website[35] and GitHub[36].

5.8 OWNCLOUD

OwnCloud is the predecessor of Nextcloud found in Section 5.5. It was founded in 2012 by a similar group of developers that founded Nextcloud in 2016. As Nextcloud is a hard fork of ownCloud the code base and idea behind it are the same with minor differences. The founder of Nextcloud mentioned the reason for doing such a fork was that "[...] there are some moral questions popping up for me."[37]. There is therefore no real difference on the technical side between both applications that is interesting to our study.

No official committer list was found for the ownCloud project. Additionally, we were able to find a mailing list for ownCloud. Further information and the code can be found on their website[38] and GitHub[39].

30 https://www.openssl.org/
31 https://www.openssl.org/community/committers.html (2020-08-20)
32 https://www.openssl.org/community/omc.html (2020-08-20)
33 https://www.openssl.org/community/otc.html (2020-08-20)
34 See footnote 31.
35 See footnote 30.
36 https://github.com/openssl/openssl
37 https://karlitschek.de/2016/04/big-changes-i-am-leaving-owncloud-inc-today/ (2020-11-04)
38 https://owncloud.com/
39 https://github.com/owncloud/core

6 EVALUATION

In this chapter, we provide a look at the obtained data about how well each network classification metric fairs against our Ground Truth. This is then discussed in detail differentiated by project and once again as a whole, answering the research questions from Chapter 1. Afterward, we discuss and explain the threats to validity.

6.1 RESULTS

We analyze the *commit data* and GitHub *issue data* from seven OSS projects found on GitHub using CODEFACE and CODEFACE-EXTRACTION. Additionally, we analyze the *mailing list data* from OpenSSL and ownCloud. Building the networks and classifying all developers into *core developer*s and *peripheral developer*s is done by CORONET through different network classification metrics. Our Ground Truth is created by analyzing the GitHub *issue data* and filtering it for events that can only be triggered by people that have at least *write* permission on the GitHub project as explained in Section 2.2. To ensure the comparability of our data, we only analyze time windows that include the full nine months (Section 4.3.3). Our scripts then compare the outputs of the network classification metrics using different measurements described in Chapter 4. In this section, we present the general results of this analysis.

6.1.1 *General Results*

6.1.1.1 *General Results concerning Directed or Undirected Networks*

Before reviewing the exact results, some general statements about the projects can be made. As can be seen for example in Figure 6.3a and Figure A.4a, undirected networks and directed networks have similar values in regards to Precision, Recall and F1. For all seven projects this holds, as the plots are similar when concerning oneself with the question of directed or undirected networks. An intuition on this is the fact that in directed networks only the users that respond to a topic get connections to the others therefore leading to an increase in connections every time they answer to a big topic. This is a big disadvantage when users that should be considered as *core developer*s give a general idea and direction to be worked on and do not get any connections to all the people they influenced or pushed in the right direction with their statement. This of course also happens to *peripheral developer*s as they also get all these connections however any *core developer* makes up for this by being active on many more topics other than this one. Since we found that there is not much of a difference between the results of directed and undirected networks, we restrict ourselves to a detailed analysis of undirected networks.

6.1.1.2 *General Results concerning Author Relations*

In the author relations, out of all seven combinations of ways to build the networks, it emerges that only a few of them are really valuable to take a look at. Cochange Networks are worth to take a look at as they describe some of the most important connections authors can have. Issue Networks are also valuable and need to be looked at as they are better on nearly all values since they are build on the same data as the Ground Truth. Out of all the combinations of all three author relations only the network built on *Issues* and *Cochange* contains interesting values. As for why no Mail Networks are taken into consideration, they either do not contain enough data to elicit a response or they are overshadowed by either *Issue* or *Cochange* or both as both of the latter contain more data and have therefore many more edges that come into play, basically taking over the majority and displaying it.

6.1.1.3 *General Results concerning Mail*

At first we took Mail Networks into account but over time we found that on the one hand very few projects continue to use mailing lists especially when GitHub is available and that even if a mailing list continues to exist there is not much discussion on it. This leads to result plots as in Figure 6.11j or Figure 6.9j. As we only found two projects that continue using mailing lists we will explain them shortly.

6.1.1.4 *General Results concerning Inclusion of Collaborators in the Ground Truth*

Another general result concerns the question, if our GT should be built with the events (see Table 3.1) contained in *elite privileges* or if we should combine both the *elite privileges* as well as the *collaborative privileges*. An analysis of all possible networks and their combinations has shown that the GT consisting only of the *elite privileges* performs slightly to clearly better in five out of seven projects. In the other two cases, the combined GT performs clearly better. It is however noteworthy, that in five out of these seven projects the combined GT has higher precision values while at the same time lower recall values. The precision values often reach 100%, meaning that all developers that are classified as such by the network classification metrics are contained within the GT. The low recall values on the other hand show that, even though this high overlap is the case, the GT got bigger as well and that now the ratio of *core developer*s that are contained within the GT, and are classified as *core developer*s by the network classification metrics, is a lot smaller. This means that even though the network classification metrics only find *core developer*s, they miss out on even more. As we weight both recall and precision the same in our study, we get the aforementioned results of five out of seven projects that perform better using the *elite privileges* GT. Hence we conclude that we further investigate the comparison to the GT created using the events in *elite privileges*.

6.1.2 *Project Results*

In the following sections, we describe and discuss all results obtained from our study. We only work with plots that have the following attributes/variable combinations:

1. *Undirected*

2. *Unsimplified*
3. Networks are built using *Cochange* or *Issue* or *Cochange and Issue* relations
4. Lines of Code and Commit Count will *not* be discussed extensively (as already stated above)
5. Any networks that do not conform to the first four points are available in the Appendix

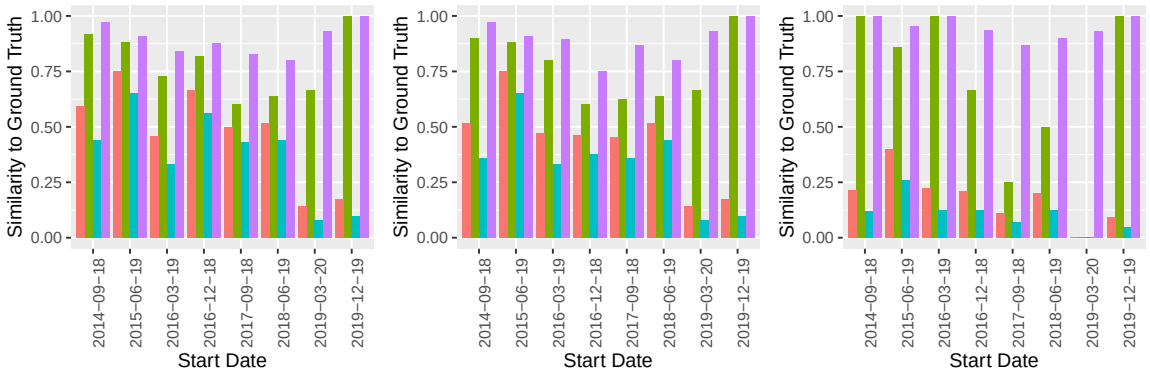
6.1.2.1 *Angular*

Angular is one of the largest projects we take a look at. It is a highly dynamic project with a high turnover, since there are on average 13 commits per author. This points to a project which has many different authors that want to participate but only implement or help with one feature. We are starting with comparing all network outputs with the Ground Truth created by us on the GitHub *issue data*.

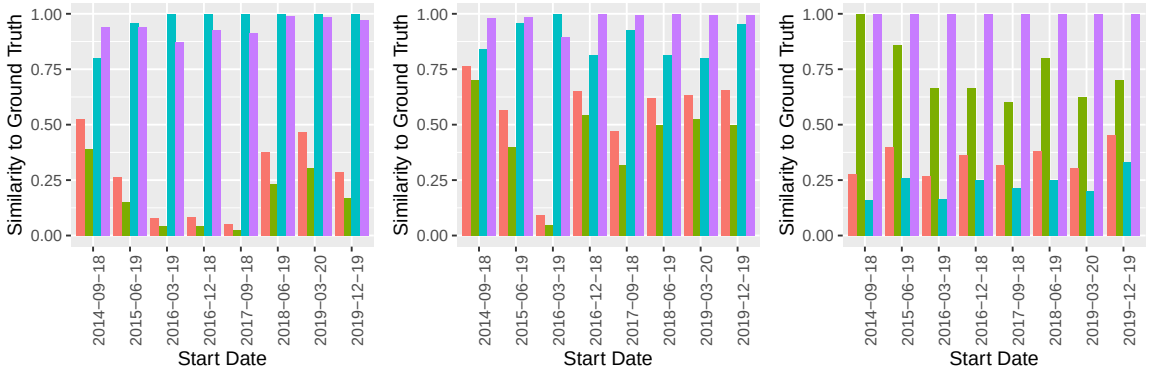
When looking at the Cochange Networks (Figure 6.1a, Figure 6.1b, Figure 6.1c), we find that both the node degree and eigenvector centrality classification look similar to each other. For both of them the precision values are high with the lowest being 65% going up to 100%. This also counts for the specificity values, although they are always slightly bigger than the precision values. The recall values are also high with most of them having numbers of over 40%. However the last two time windows contain only very small values. This also means that the F1 score is overall above 40% except for these last two time windows. The hierarchy classification has on three of its time windows precision values of 100% while at the remaining time windows they exceed 50% three times. The recall values however are very low with a maximum of 25%.

A different picture can be seen on the Issue Networks (Figure 6.1d, Figure 6.1e, Figure 6.1f) where the precision values for the hierarchy classification are the highest while they are the lowest in the node degree classification. In the node degree classification all precision values never cross 40% and there are even three time windows in the middle where the values do not exceed 10%. On the other hand the recall values are all very high with the minimum being at 80% while most of them are at 100%. The eigenvector centrality classification has similar recall values and even though their minimum is also 80% this minimum is hit for times as often as in the node degree classification. The precision values are a lot higher as they are mostly in the range between 40% and 70% with one time window having a value below 5%. This leads to an F1 score that is in all but one time window close to or above 50%. The hierarchy classification on the other hand generally has good precision values that are consistently over 60%. The recall values on the other hand are never bigger than 30% and have a relatively stable number. As we can see, even though there are not many issues that can be analyzed in the later stages of the project, they still coincide with our issue-based Ground Truth.

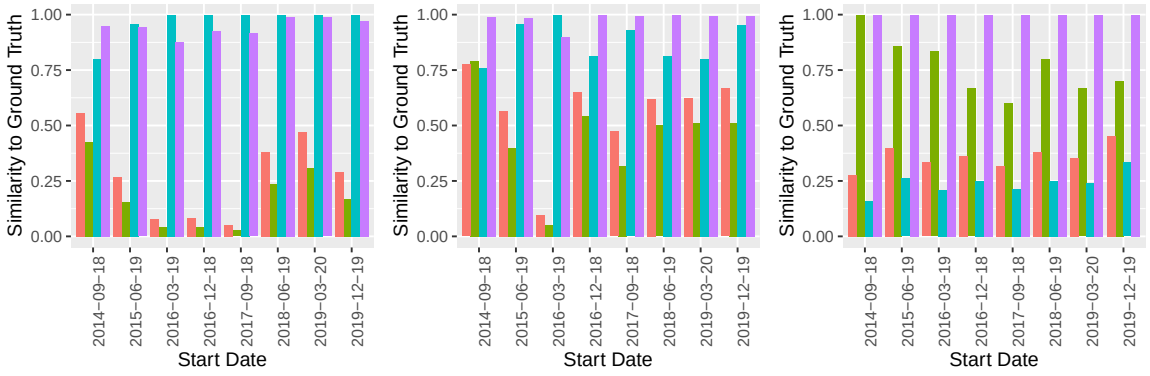
As for a combination of both the Issue Networks and the Cochange Networks it is clear that the Issue Networks are making up the main data as the plots of Issue Networks (Figure 6.1d, Figure 6.1e, Figure 6.1f) and the Cochange Issue Networks (Figure 6.1g, Figure 6.1h, Figure 6.1i) look very much alike. On the other hand the Cochange Issue Networks achieve even higher values than the Issue Networks in some time windows, for



(a) Node Degree on Cochange Network (b) Eigenvector Centrality on Cochange Network (c) Hierarchy on Cochange Network



(d) Node Degree on Issue Network (e) Eigenvector Centrality on Issue Network (f) Hierarchy on Issue Network



(g) Node Degree on Cochange Issue Network (h) Eigenvector Centrality on Cochange Issue Network (i) Hierarchy on Cochange Issue Network

(j) Legend

Figure 6.1: Angular - All undirected networks that are compared to the issue-based Ground Truth.

example the first one, suggesting that not all of the *core developer*s can be found through Issue Networks only, as they might only have worked on the code in this time window rather interacting with other developers on GitHub.

As a general result for Angular, it can be seen that the eigenvector centrality classification metric performs best on Cochange Issue Networks.

RQ1-Angular: Eigenvector Centrality on Cochange Issue Networks performs best

Similar results to the one above are also seen when comparing the networks with the official list.

A clear disadvantage of the committer Ground Truth taken from the official committer list is the timing of whom this committer list includes. Looking at all the different networks and metric combinations in Figure A.2, we can see that the values are generally increasing to the end and that mostly the highest values are in the last time window. Furthermore the clearest increase can be seen when looking at the eigenvector centrality classification on the Cochange Issue Network (Figure A.2h), where the increase can be seen to be basically linear.

But it is also apparent that again the Cochange Issue Networks return the best results with eigenvector centrality being the classification metric.

RQ2-Angular: Eigenvector Centrality on Cochange Issue Networks performs best

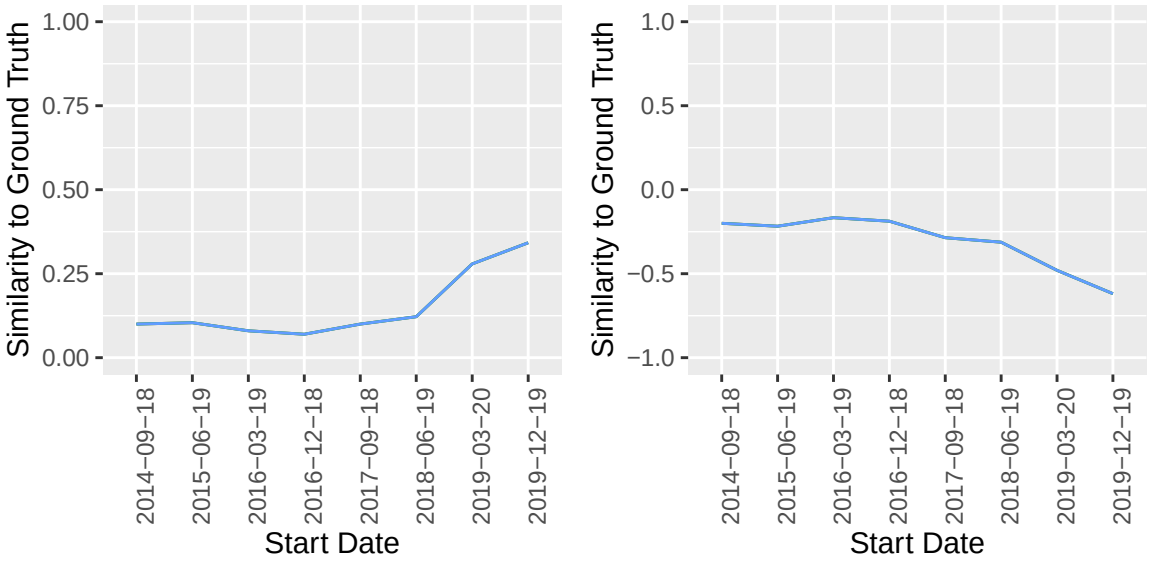
Comparing both the Ground Truths leads us to an expected plot. Starting with only some amount of the Jaccard similarity between the two, this value increases up to 40% in the most recent time windows. Looking at the Overlap value for this suggests that with close to 70% our Ground Truth is contained within the official committer list.

RQ3-Angular: Both of the Ground Truths overlap and have similar elements, although the official committer list finds more people than our Ground Truth.

6.1.2.2 *Data Transfer Project*

In contrast to Angular the Data Transfer Project is a more stable project without as much of a turnover since every commit author is on average responsible for 38 commits in the time working for the project. We again start analyzing the networks compared to our Ground Truth. A committer list was not found and RQ2 and RQ3 can therefore not be answered or considered for this project.

When taking a look at the classification plots for the Cochange Networks (Figure 6.3a, Figure 6.3b, Figure 6.3c), the precision values for all time windows are between 75% and 100%. In the hierarchy classification they are even always 100%. The recall values in both the node degree and eigenvector centrality classifications are in the range of 40% and 55% while in the hierarchy classification they never pass 40%. This leads to a higher F1 score in the node degree and eigenvector centrality classifications with values up to 70%. Noteworthy is, that in the node degree and eigenvector centrality classifications the time window 2018-10-05 does not include a bar for the specificity value. The reason for this is that all people that were identified to be *peripheral developer*s by the network classification



(a) Jaccard Values

(b) Overlap Values

Figure 6.2: Comparison of both Ground Truths using Jaccard and Overlap. Negative Overlap values indicate that the official committer list was bigger.

metrics were identified by us to be part of the issue-based Ground Truth. This means that there are no true negatives and this leads to a value of 0% for specificity.

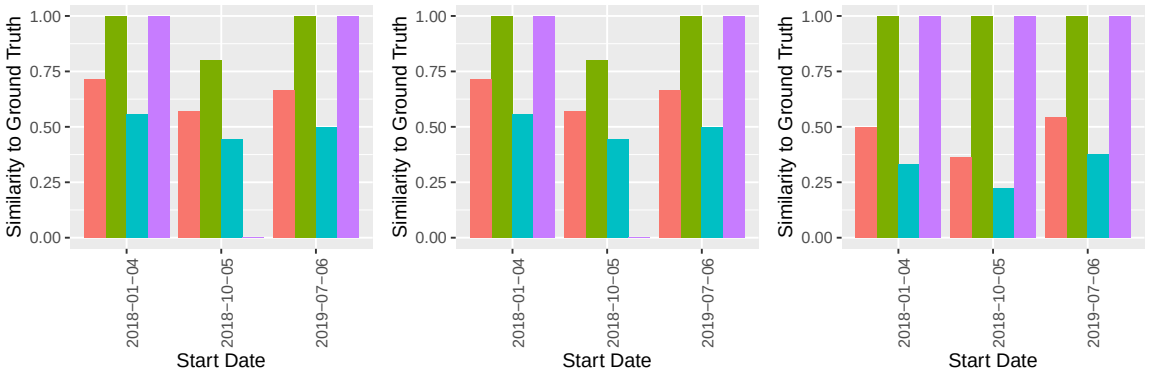
In the Issue Networks (Figure 6.3d, Figure 6.3e, Figure 6.3f) a slightly different picture is seen as the precision values are at 100% at all time windows for all centrality metrics. This is also true for the specificity values. The only difference is the recall values and for the hierarchy classification these values are all below 40%. For the other network classification metrics, they are basically the same values being in the range of 40% to 70%.

The combination of both the Cochange and Issue Networks is again an indicator that the Issue Network has way more data than the Cochange Network as the Issue Network dominates. The Cochange Issue Network (Figure 6.3g, Figure 6.3h, Figure 6.3i) has therefore basically the same values as the Issue Network. Only at some points like in the first time window of Figure 6.3g does the Cochange Network add users to the *core developer* list that were not there before, helping to increase the scores.

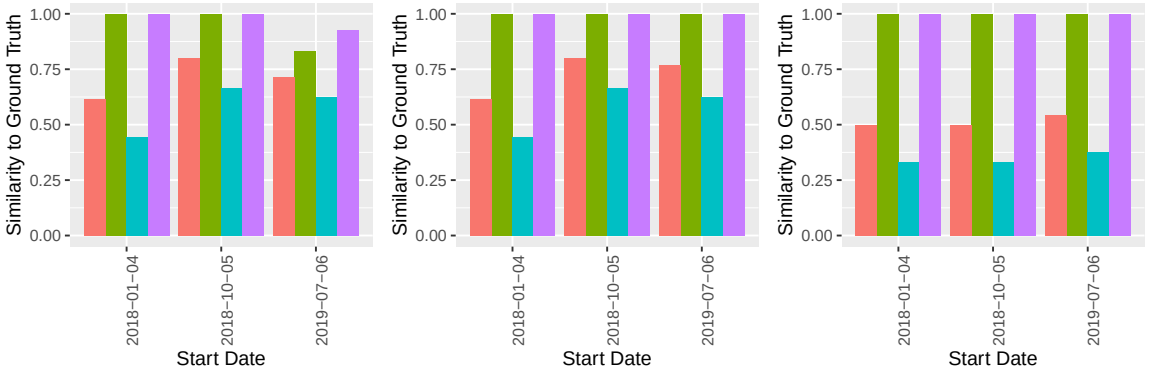
In a general result for the Data Transfer Project, it can be seen that the node degree and eigenvector centrality classifications on the cochange and issue combination are the strongest while the corresponding metrics in the Issue Network are only as good as the former.

> *RQ1-DTP: Node Degree and Eigenvector Centrality on Cochange Issue Networks perform best*

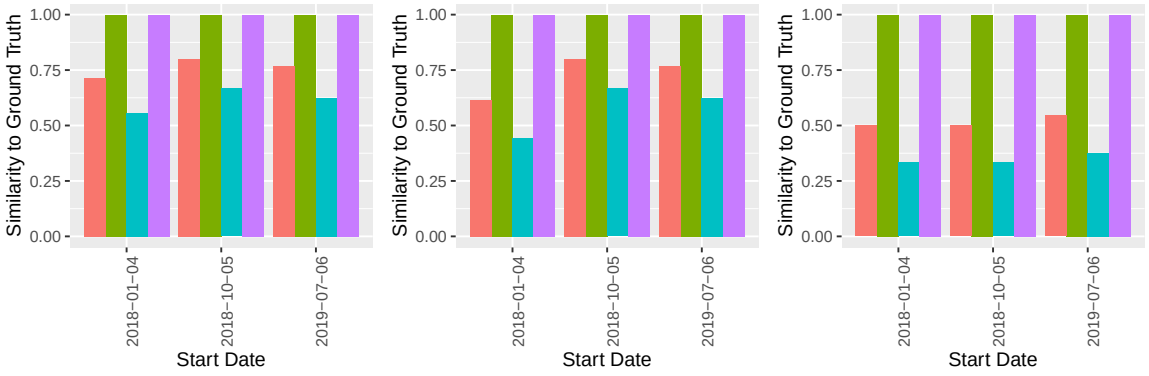
As mentioned before RQ2 and RQ3 can not be answered as no committer list was found for the project.



(a) Node Degree on Cochange Network (b) Eigenvector Centrality on Cochange Network (c) Hierarchy on Cochange Network



(d) Node Degree on Issue Network (e) Eigenvector Centrality on Issue Network (f) Hierarchy on Issue Network



(g) Node Degree on Cochange Issue Network (h) Eigenvector Centrality on Cochange Issue Network (i) Hierarchy on Cochange Issue Network

(j) Legend

Figure 6.3: Data Transfer Project - All undirected networks that are compared to the issue-based Ground Truth.

6.1.2.3 *Keras*

Keras has become a unique case for our study, as can be seen in Figure 5.1 where the number of lines over time can be seen. The reason for the drop from more than 80.000 lines of code to a mere 521 at the current time window is the fact that the Keras library was integrated into TensorFlow and now resides there while the current Keras project is only an API for a call to the Keras within TensorFlow. Additionally, Keras is a project with a high turnover as only six commits are on average done by a commit author.

Starting with the Cochange Networks (Figure 6.4a, Figure 6.4b, Figure 6.4c), only the hierarchy classification has a precision at 100% at multiple time windows. In the other plots displaying node degree and eigenvector centrality, the precision values are always lower than their respective recall values. What is noteworthy is the hierarchy classification as it produces very high precision values while having small recall values as opposed to both being at around 25% to 50% as can be seen with the other network classification metrics.

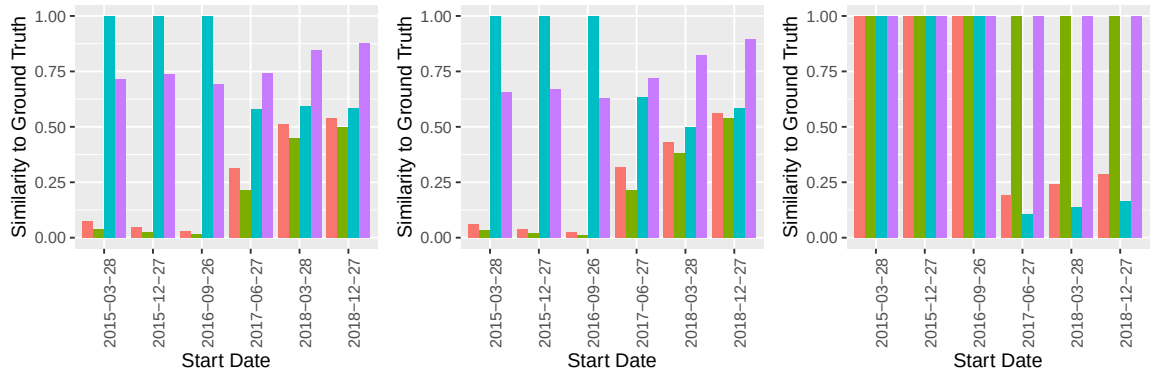
This repeats itself when looking at the Issue Networks (Figure 6.4d, Figure 6.4e, Figure 6.4f), as precision is very low if existent while the recall values are uncharacteristically high, some even meeting 100%. The F1 score is in general low while there are some outliers especially at the end of node degree and in the eigenvector centrality. The hierarchy classification again has high precision and sometimes even recall values which lead to an F1 score of about 30% to 40% in the last time windows. This change from precision having the highest values to it being overshadowed by recall comes from the fact that there was a very high activity in the Issue Networks with approximately 3.000 users taking part in the discussion and or asking questions. This is definitely an outlier, since in other projects there is an activity of maybe around 500 people overall on the Issue Network in one time window, however here about 600 people are classified as *core developer*s for their activity.

This picture paints itself again in the combination of the networks (Figure 6.4g, Figure 6.4h, Figure 6.4i) where again the Issue Networks make up the biggest part leading to very much the same plots as before. Only in the hierarchy network (Figure 6.4i) can we see a definite influence of the Cochange Network on the data, as the first three time windows have all values at 100% again.

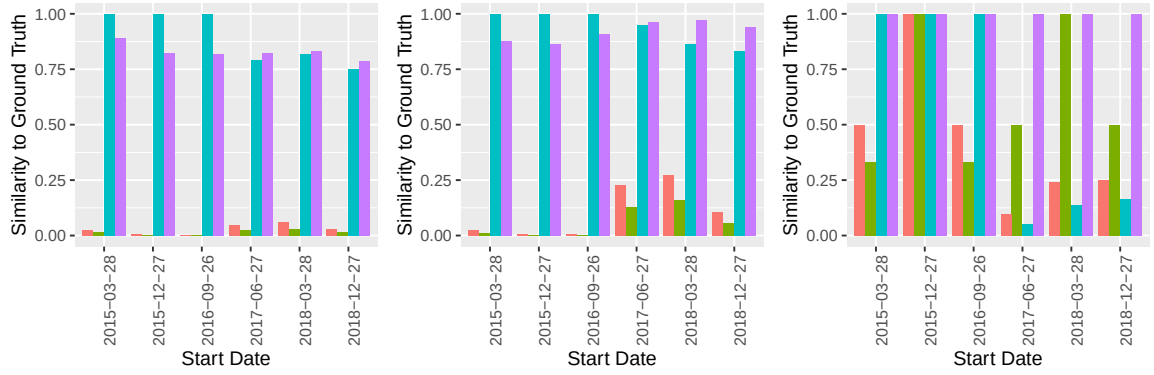
RQ1-Keras can not be as easily answered as before as some network classification metrics lead to a high precision while others lead to a very high recall. We will therefore be taking the F1 score as the most important measure. Additionally, as before, the combination of both Cochange Networks and Issue Networks performs best.

RQ1-Keras: Hierarchy on Cochange Issue Networks performs best

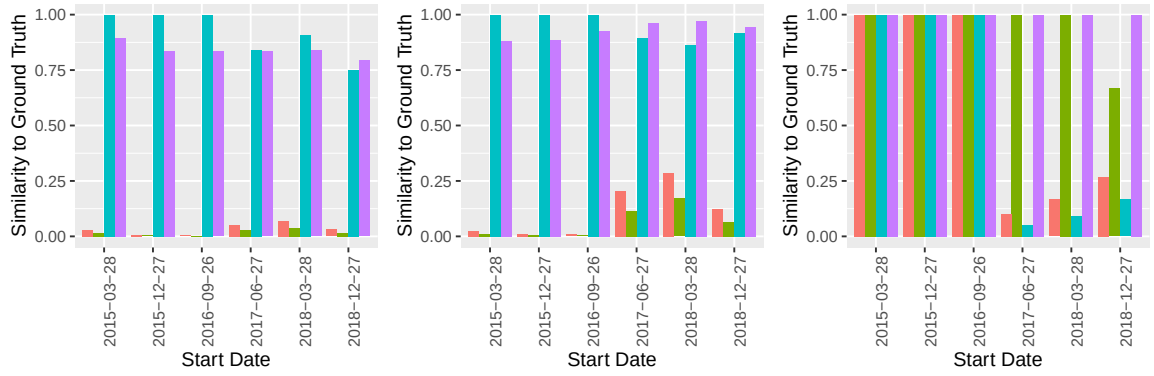
Taking a look at the results in Figure A.8 generated with the official committer list, three things are apparent. First, the Ground Truth does not work for the earlier stages of the project, as we can see no precision or recall values earlier than September 2016 which are still very low. This is the case for all generated plots. Second, the values of precision, recall and therefore the F1 score are all very low. The recall values can be seen to be around 40% on the node degree and eigenvector centrality classification in the later time windows. The precision values on the other hand are basically negligible. Three, the only metric which actually does result in some values for the F1 score is the hierarchy classification. It shows



(a) Node Degree on Cochange Network (b) Eigenvector Centrality on Cochange Network (c) Hierarchy on Cochange Network



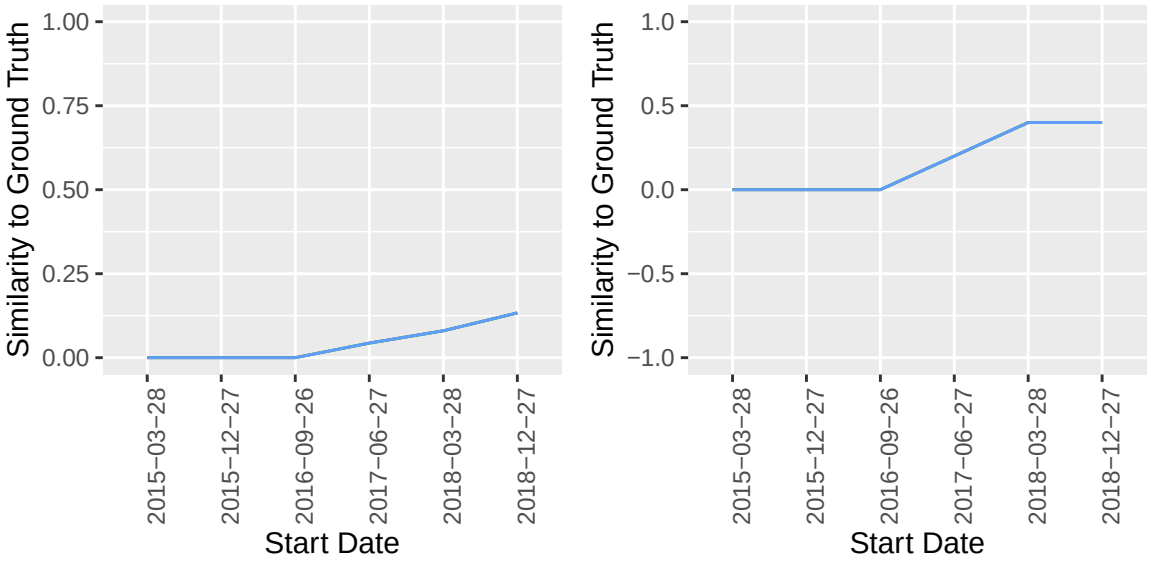
(d) Node Degree on Issue Network (e) Eigenvector Centrality on Issue Network (f) Hierarchy on Issue Network



(g) Node Degree on Cochange Issue Network (h) Eigenvector Centrality on Cochange Issue Network (i) Hierarchy on Cochange Issue Network

(j) Legend

Figure 6.4: Keras - All undirected networks that are compared to the issue-based Ground Truth.



(a) Jaccard Values

(b) Overlap Values

Figure 6.5: Comparison of both Ground Truths using Jaccard and Overlap. Negative Overlap values indicate that the official committer list was bigger.

values of recall and precision being around 25% to 30% which means the F1 score is also in that vicinity. This is especially so for the Issue and Cochange Issue Networks. Here again the high amount of people that partook in the GitHub issues can be seen, however the values for recall are not as high in comparison to networks that were analyzed using our Ground Truth. All in all, the same answer from RQ1-Keras stands for RQ2-Keras.

RQ2-Keras: Hierarchy on Cochange Issue Networks performs best

Comparing both Ground Truths we can see that the Jaccard value between the two is not good and does not exceed 25%. This in conjunction with a maximum overlap value of 50% leads us to accept that the Keras committer list is as previously mentioned a group of people that guide the project but do not actively take part in it. This is especially shown in the fact that the Keras committer list only includes five developers. It is therefore not really a committer list and a comparison can not be done.

RQ3-Keras: Can not be compared, due to the official committer list being a project leader list.

6.1.2.4 *Nextcloud*

Nextcloud is one of the biggest projects analyzed. It also the project with the second-highest number of commits per commit author, on average about 63. This points to a very low turnover. As previously mentioned, Nextcloud is also a company that has made its code Open-Source Software. This coincides with the many commits per commit author, as these authors are employed and working at the project as a job. This count is therefore no surprise. When looking at the plots it is important to note that while the first GitHub activity

mentioned in Table 5.1 is 2010-03-10 this company does only exist since 2016. All reported GitHub activity is there because this project was a fork of ownCloud (5.8). All our plots are, however, only analyzing the situation and dynamics since 2016-06-02. Analyzing the plots, we will start again with the plots concerning the issue-based Ground Truth and later taking a look at the committer list. An official committer list was not found and RQ2 and RQ3 can therefore not be answered or considered for this project.

Starting with the Cochange Networks (Figure 6.6a, Figure 6.6b, Figure 6.6c) compared to our issue-based Ground Truth, we can see a big difference between the values of hierarchy and the other two network metrics. In the hierarchy network the precision and specificity values are consistently at 100% while the recall values are rather low sitting at a maximum of 25%. On the other hand, both the node degree and eigenvector centrality metrics have high values in both specificity and precision as well as in recall. The specificity values are consistently at 100% and the precision values are always between 75% and 100%. Additionally, the recall values can be seen to be around 35% to 60%.

This feat of high precision and high recall values can not be repeated by the Issue Networks (Figure 6.6d, Figure 6.6e, Figure 6.6f). When analyzing the hierarchy classifications, all specificity and precision values are very similar to the ones seen in the Cochange Networks. Interesting is that the values of recall are very stable compared to before since the values range between 15% and 20%. For eigenvector centrality and node degree classification the precision values are not high as the maximum sits at 70% and 50% respectively, in the first time window with the rest of the values declining. In contrast to this, the recall values in the eigenvector centrality classification are really high ranging from 70% to approximately 90%. The recall values in the node degree classification are similarly high however not as high as in eigenvector centrality with the lowest value being 50% and the highest 90%.

In the Cochange Issue Networks (Figure 6.6g, Figure 6.6h, Figure 6.6i) the pattern of dominating GitHub *issue data* is repeated as there is no visible difference between them both. This leads to the Cochange Network having clearly better F1 scores since these are sometimes twice as high as the respective values in the issue and Cochange Issue Networks.

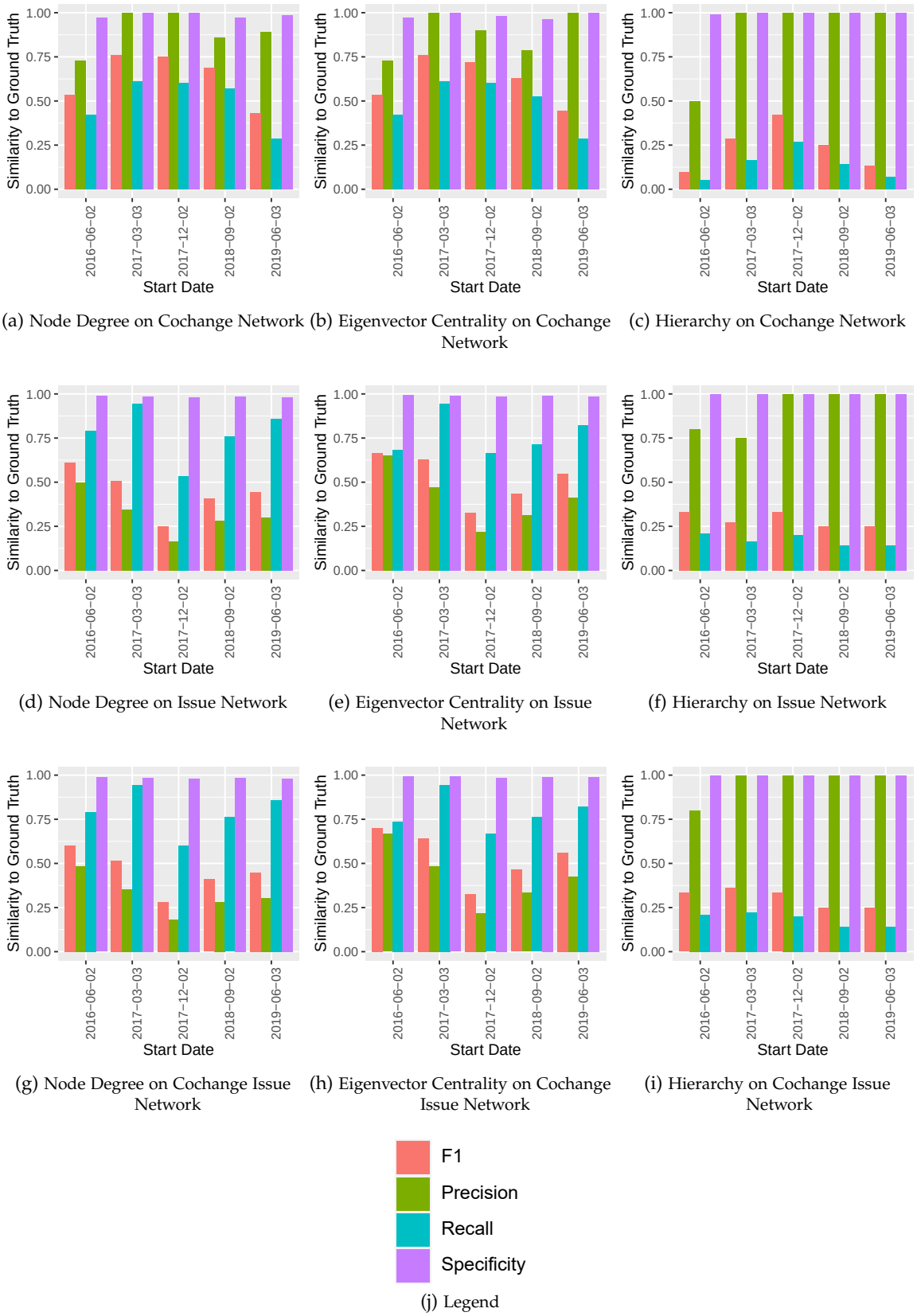
> *RQ1-Nextcloud: Node Degree and Eigenvector Centrality on Cochange Networks perform best*

As mentioned before RQ2 and RQ3 can not be answered as no committer list was found for the project.

6.1.2.5 *NodeJS*

NodeJS is the biggest project with its 9.5 Million lines of code; that we take a look at. It is also the project with most people that authored commits. Due to having a relatively (to its number of lines) low number of commits, any developer has on average ten commits. This points again to a project with a relatively high turnover in developers. We will start by looking at the networks compared against our issue-based GT and continue by taking a look at the networks compared against the official committer list.

Looking at the Cochange Networks (Figure 6.7a, Figure 6.7b, Figure 6.7c) first, we can see that in the node degree and eigenvector centrality classifications, the values for precision



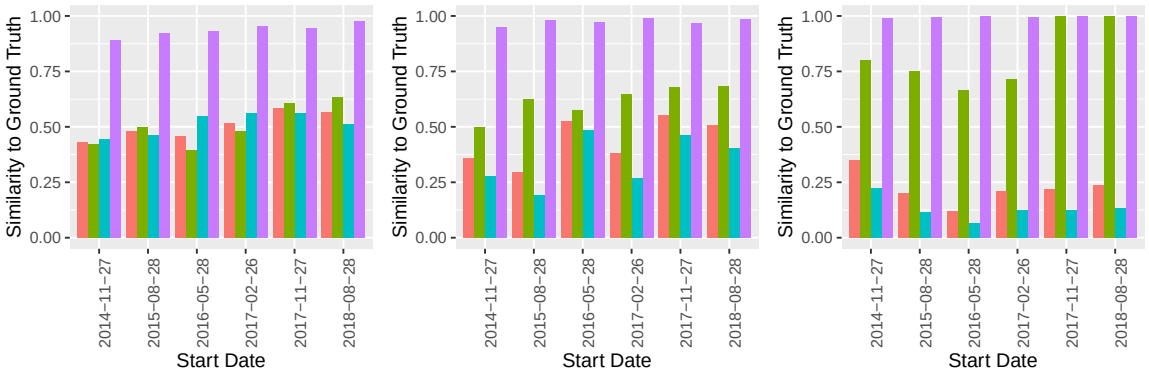
(a) Node Degree on Cochange Network (b) Eigenvector Centrality on Cochange Network (c) Hierarchy on Cochange Network

(d) Node Degree on Issue Network (e) Eigenvector Centrality on Issue Network (f) Hierarchy on Issue Network

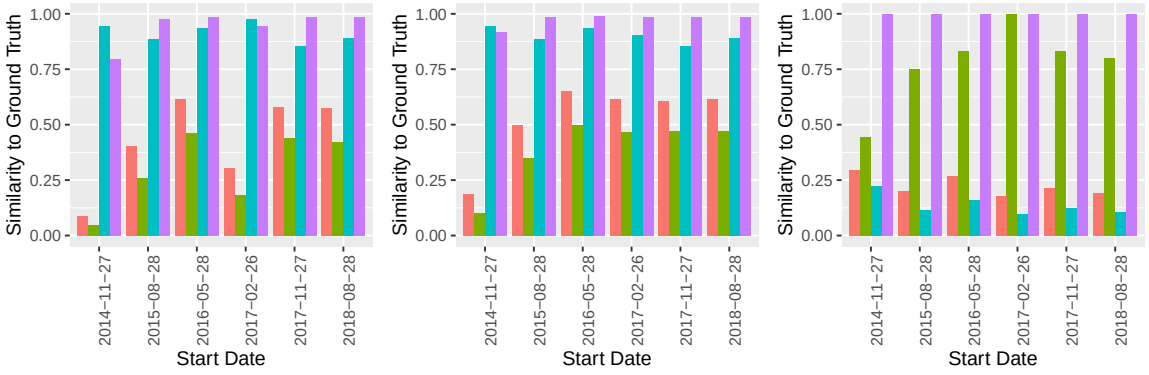
(g) Node Degree on Cochange Issue Network (h) Eigenvector Centrality on Cochange Issue Network (i) Hierarchy on Cochange Issue Network

(j) Legend

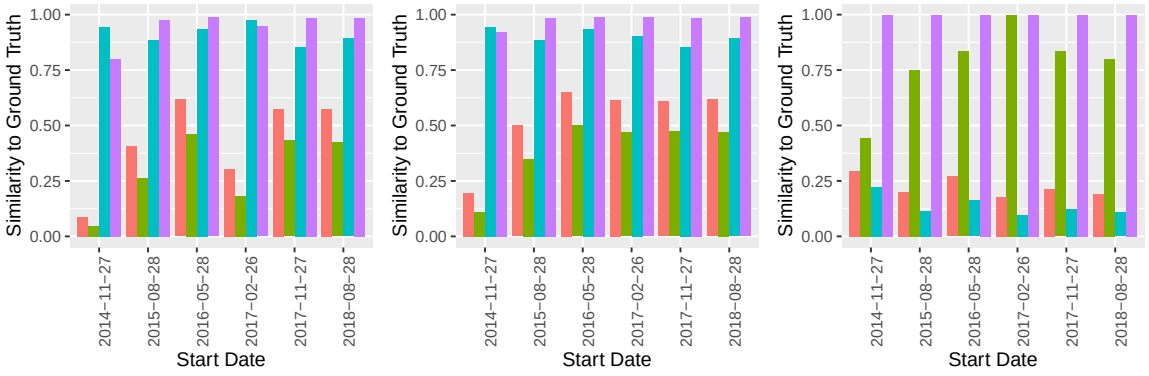
Figure 6.6: Nextcloud - All undirected networks that are compared to the issue-based Ground Truth.



(a) Node Degree on Cochange Network (b) Eigenvector Centrality on Cochange Network (c) Hierarchy on Cochange Network



(d) Node Degree on Issue Network (e) Eigenvector Centrality on Issue Network (f) Hierarchy on Issue Network



(g) Node Degree on Cochange Issue Network (h) Eigenvector Centrality on Cochange Issue Network (i) Hierarchy on Cochange Issue Network

(j) Legend

Figure 6.7: NodeJS - All undirected networks that are compared to the issue-based Ground Truth.

are in the range of 40% to 70%. The recall values in the node degree classification are all very stable as they range from 40% to 60%. On the other hand, in the eigenvector centrality classification the recall values are slighlty lower as they range from 20% to 50%. The specificity values for all plots are high and reach the range of 85% to 100%. Analyzing the hierarchy classification, precision is generally higher as these values are all between 70% to 100%. Recall however is way smaller with values ranging from 5% to 25%. This also leads to rather small values of the F1 score as this never exceeds 30%.

For the Issue Network (Figure 6.7d, Figure 6.7e, Figure 6.7f), the precision values are slightly lower in all plots. In the node degree classification they do not pass 50%, having drops in value in the beginning and the end. Contrary to this, the values are smooth in the eigenvector centrality classification as the middle drop is missing and the values look like a curve. For the hierarchy classification all precision values are bigger as they also build a curve however this one reaches 100% as its maximum value. The recall values are especially high for both the node degree as well as the eigenvector centrality classification. They are situated in a range from 80% to over 90%. This also leads to a high F1 score at around the 50% mark. Only in the hierarchy classification can we see that the recall values are very low as they are never above 25%. Concerning the values of specificity, they are always nearly at 100%.

In the combination of both network types (Figure 6.7g, Figure 6.7h, Figure 6.7i) as previously mentioned, the Issue Networks are dominating and there is no obvious difference between the Issue Network and the Cochange Issue Network analysis.

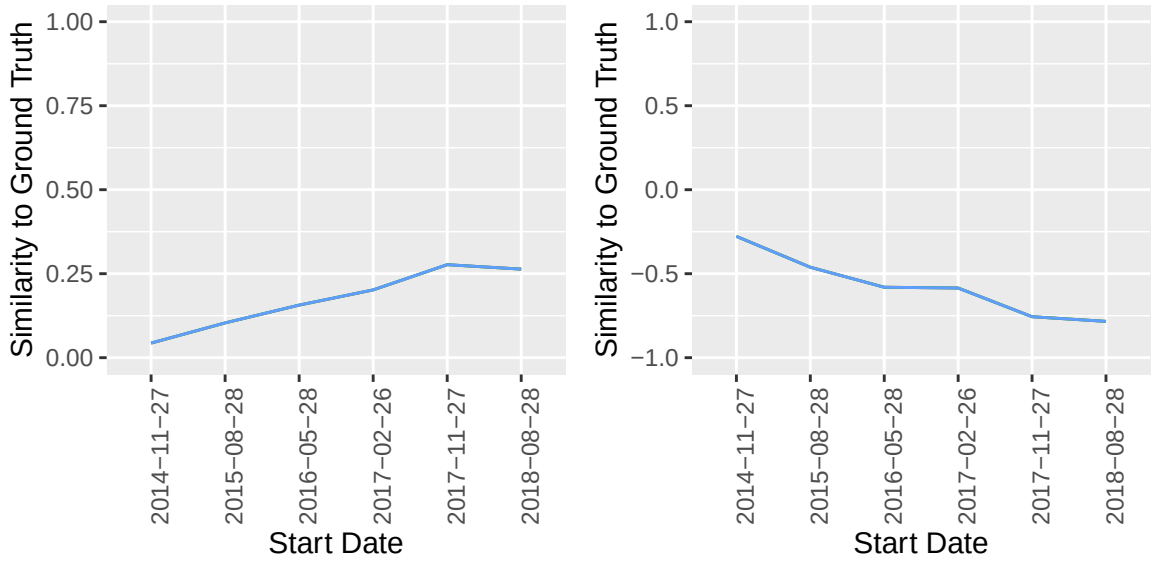
RQ1-NodeJS is easily answered as the eigenvector centrality metric has consistently, except for one time window, higher values. Furthermore, the Issue Networks and the Cochange Issue Networks have the same values.

RQ1-NodeJS: Eigenvector Centrality on Cochange Issue Networks performs best

Taking a look at the networks compared with the official committer list in Figure A.13, we can see a similar picture, as the precision values are mostly above 50% and the specificity values at 100%. The precision values are slightly higher than in the networks compared with the issue-based Ground Truth, however the recall values are rather small, never passing 25% in the Cochange Network. In the Issue Network, this trend of small recall values is not the case as they are rising to the end to about 50%. We therefore also get high F1 scores on the Issue Network. The only exception in both the Cochange Network and Issue Network is the hierarchy classification. Its precision values are high, mostly being above 50% often even being above 75%. Contrary to these high values, recall is always below 10%. Furthermore, the Issue Networks generally have higher values than the Cochange Networks and are showing the expected rise to the end. As the Cochange Issue Networks are again dominated by the Issue Networks, the eigenvector centrality classification on Cochange Issue Networks can be considered best.

RQ2-NodeJS: Eigenvector Centrality on Cochange Issue Networks performs best

Comparing the two Ground Truths using the Jaccard similarity, we can then see the expected rise at the end to about 25%. However when looking at the Overlap similarity we can see that nearly all of the developers in the Ground Truth are found in the official committer list



(a) Jaccard Values

(b) Overlap Values

Figure 6.8: Comparison of both Ground Truths using Jaccard and Overlap. Negative Overlap values indicate that the official committer list was bigger.

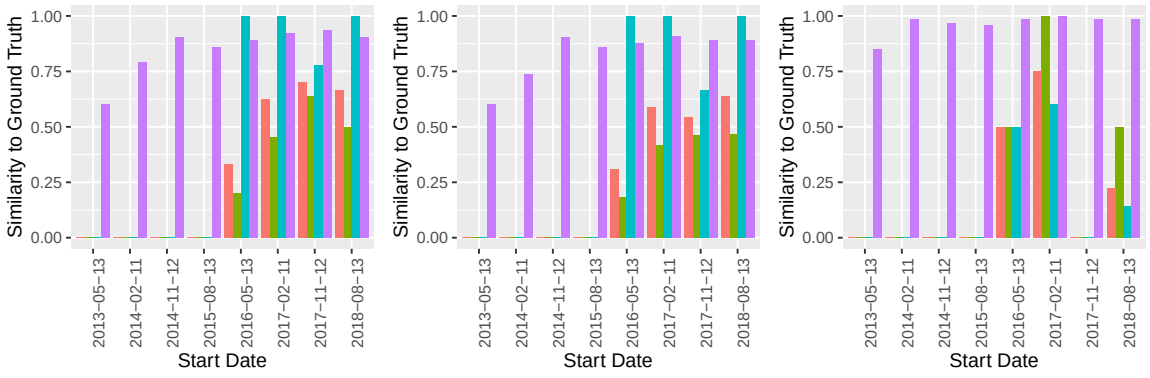
as approximately 75% of our Ground Truth are in both. This again suggests that not all *core developer*s take part within the project or are falsely on this list.

> *RQ3-NodeJS: Both of the Ground Truths overlap and have similar elements, although the official committer list contains more people than our Ground Truth*

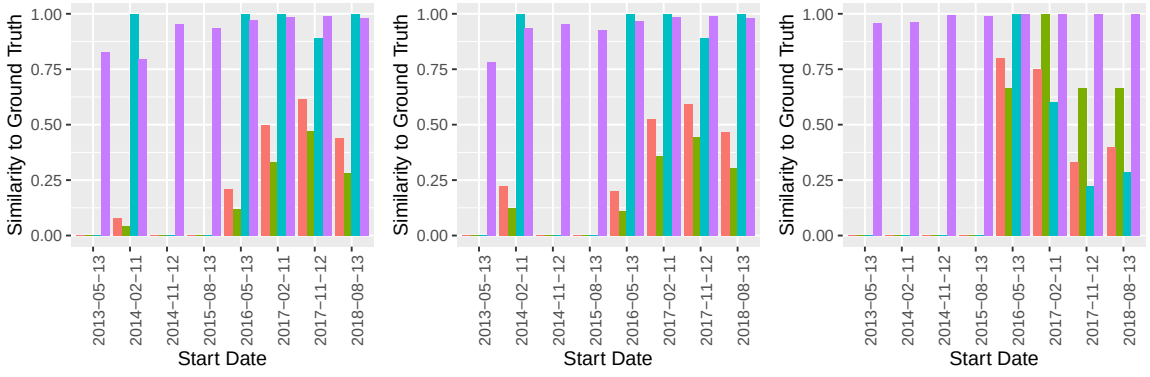
6.1.2.6 *OpenSSL*

OpenSSL is another project that has over 1 million lines of code. It is by far the oldest project, although we are not able to use this data since they were not on GitHub before but rather used other version control systems. Concerning this our plots will start in 2013 as this was the time when the first commits were done to GitHub, however the true data starts between 2015 and 2016 as this was the time of the official migration to GitHub. With around 41 commits per author OpenSSL is one of the projects with the highest amount of commits per author. Starting with the networks compared against our issue-based GT, and then going over to the official committer list, we will be able to answer all three research questions.

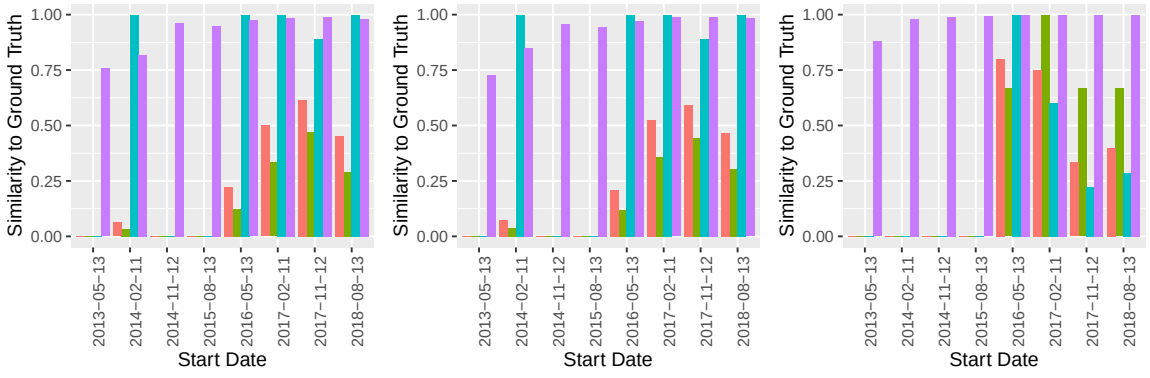
As can be seen in the Cochange Networks (Figure 6.9a, Figure 6.9b, Figure 6.9c), the first four time windows contain only values of specificity as all other values in one way or the other depend on a computation of the *core developer*s through the network classification metrics. As there is little to no data in the *commit data* during these time windows, the results show 0% for these values. When taking a look at the precision values for both the node degree and eigenvector centrality, we can see that they are rising and getting better over time as they reach up to 75% in the last time window while starting at around 20%. The recall values on the other hand do not consistently increase over time but are at 100%



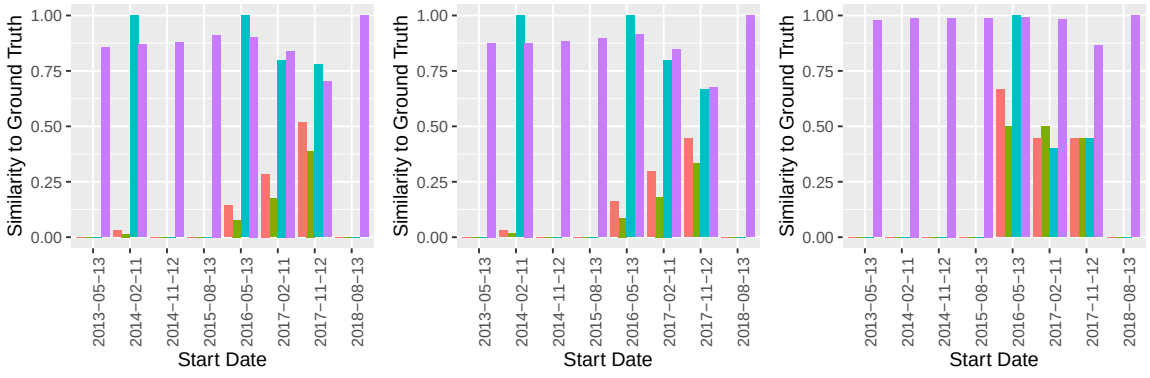
(a) Node Degree on Cochange Network (b) Eigenvector Centrality on Cochange Network (c) Hierarchy on Cochange Network



(d) Node Degree on Issue Network (e) Eigenvector Centrality on Issue Network (f) Hierarchy on Issue Network



(g) Node Degree on Cochange Issue Network (h) Eigenvector Centrality on Cochange Issue Network (i) Hierarchy on Cochange Issue Network



(j) Node Degree on Mail Network (k) Eigenvector Centrality on Mail Network (l) Hierarchy on Mail Network

Figure 6.9: OpenSSL - All undirected networks that are compared to the issue-based Ground Truth. A legend can be found in Figure 6.7.

at three out of four time windows. The other times the values are set at 75% in the node degree classifications while they are set at slightly over 65% in the eigenvector centrality classifications. In the hierarchy classification, this looks a lot different as the hierarchy could not find any *core developer*s in the time window of November 2017. Precision values are generally at 100%, but once in the first time window the value is set at 50%. Recall values are in the range of 25% to 65% while on average being around 50%.

On the Issue Networks (Figure 6.9d, Figure 6.9e, Figure 6.9f) a similar picture is seen. Noteworthy is the fact that there were some issues in 2014, which means that data is found in this one time window returning a value. But other than a high recall value of 100% all other values are negligible. On the node degree and eigenvector centrality classifications there is a rise of the precision values which ends in the last time window. In both plots these values are not higher than 50%. The recall values are consistently at 100% with one exception being at 90%. In the hierarchy, this is flipped as the precision values are mostly at 100%, while the recall values start at 100% and end with 25%.

Another important note is that for this project there was a mailing list, which leads us to have mailing data. When analyzing them we can however see that the mailing list was pretty much discontinued in 2018 as there is little to no data from that date on. Additionally, we can see a steady decline in all values from recall to specificity except for precision. We can therefore continue to analyze as if the *mailing list data* did not exist.

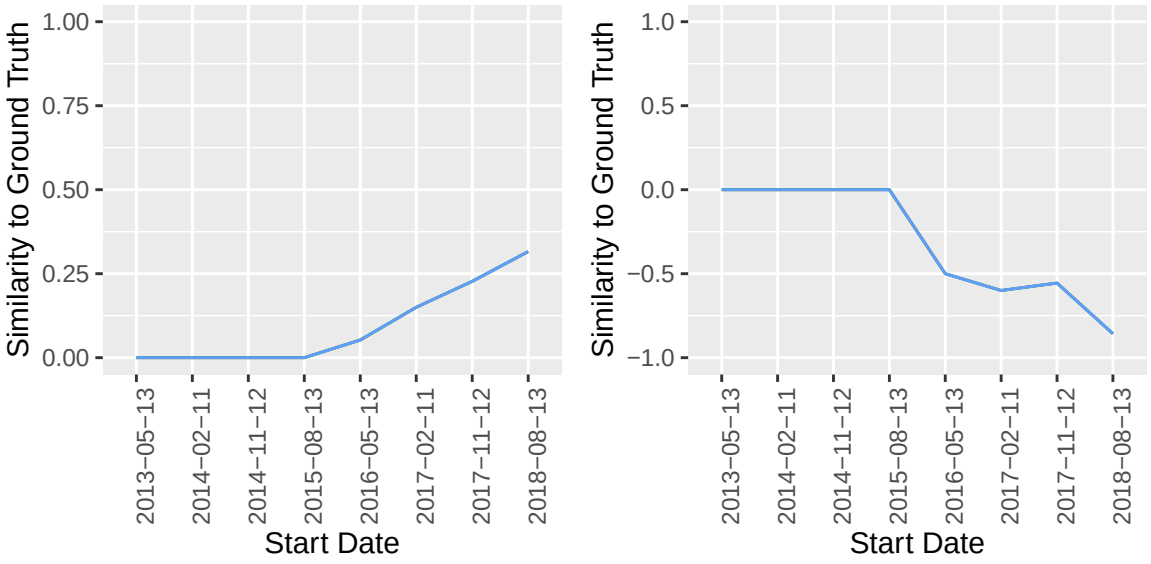
The analysis for the Cochange Issue Networks (Figure 6.9g, Figure 6.9h, Figure 6.9i) is the same as with the Issue Networks as the data is again dominated by the issues.

RQ1-OpenSSL: Node Degree on Cochange Networks performs best

As there is an official committer list against which the networks (Figure A.16) can be compared, we now have data for all time windows except for the first one. The node degree and eigenvector centrality classifications are very similar as they both follow the same curve. The precision values for both increase to over 50% in the last time window and the recall values are all between 25% and 50%. This leads to the F1 score being 25% to 50%. This is not reflected in the hierarchy classifications as precision has values rising to 100% while recall is never rising above 15%.

When looking at the Issue Networks (Figure A.16d, Figure A.16e, Figure A.16f), we can see similar precision values on both the node degree and eigenvector centrality classifications while the recall values are considerably higher especially in the right half of the graph. Here the values rise nearly linearly up to 65%. In the hierarchy classifications this trend of rising recall values is similar to the one in the Cochange Networks as the values are never more than 5% from each other. The precision values however are way higher with values of 65% and two times even 100%.

When looking at the combination of Cochange Networks and Issue Networks there are some little changes. This especially takes place in the earlier time windows, where values get better compared to the Issue Network and start to look like the Cochange Network. This happens because as mentioned before, there is not much GitHub *issue data* to be analyzed while there is some *commit data*. This leads to the *commit data* dominating the GitHub *issue data* in the earlier time windows while the GitHub *issue data* wins back its dominating



(a) Jaccard Values

(b) Overlap Values

Figure 6.10: Comparison of both Ground Truths using Jaccard and Overlap. Negative Overlap values indicate that the official committer list was bigger.

size in the later stages. The data therefore looks more like the Cochange Networks in the beginning and the Issue Networks in the end.

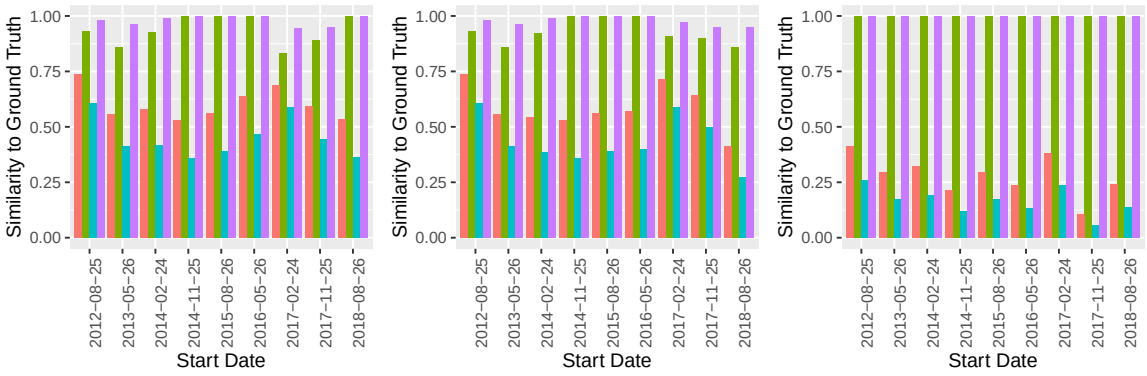
> *RQ2-OpenSSL: Node Degree and Eigenvector Centrality on Cochange Issue Networks perform best*

When comparing the official committer list against our issue-based Ground Truth, the Jaccard values are rising up to approximately 40%. Adding to this is that the Overlap values are rising nearly up to 80%. This means that basically 80% of our GT is also contained within the official committer list.

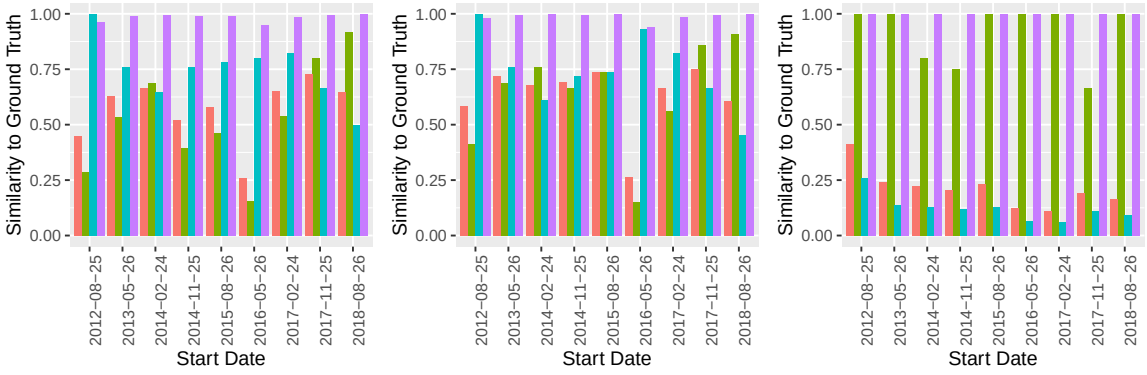
> *RQ3-OpenSSL: Both of the Ground Truths overlap and have similar elements, contains more people than our Ground Truth*

6.1.2.7 *ownCloud*

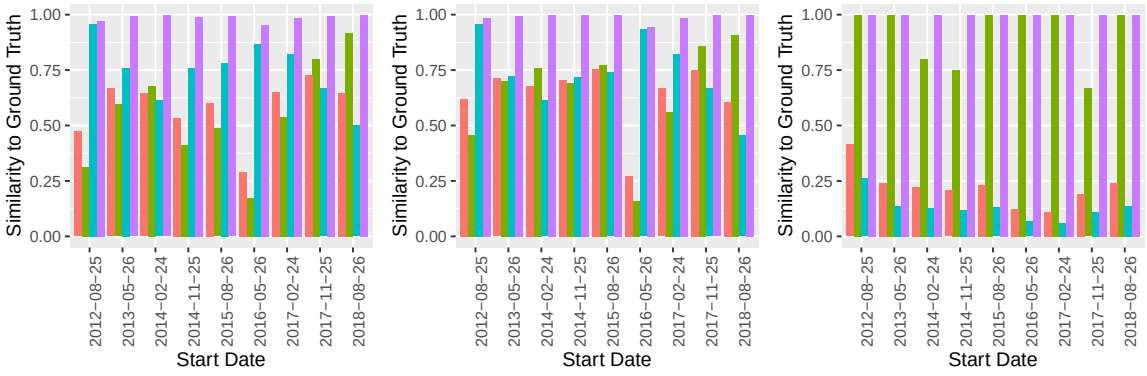
ownCloud is the second biggest Open-Source Software project we take a look at with its 1.9 Million lines of code. It is also the project with the most commits per author 72. This points to a very low turnover within the project and coincides with the fact that there is a company behind ownCloud which releases updates regularly. Even though Nextcloud and ownCloud are both working on the same base code from 2016, ownCloud has nearly 70% more lines of code in its repository than Nextcloud. This happened just recently as an update was pushed to the master branch adding more than 800.000 lines of code. An official committer list was not found and RQ2 and RQ3 can therefore not be answered or considered for this project.



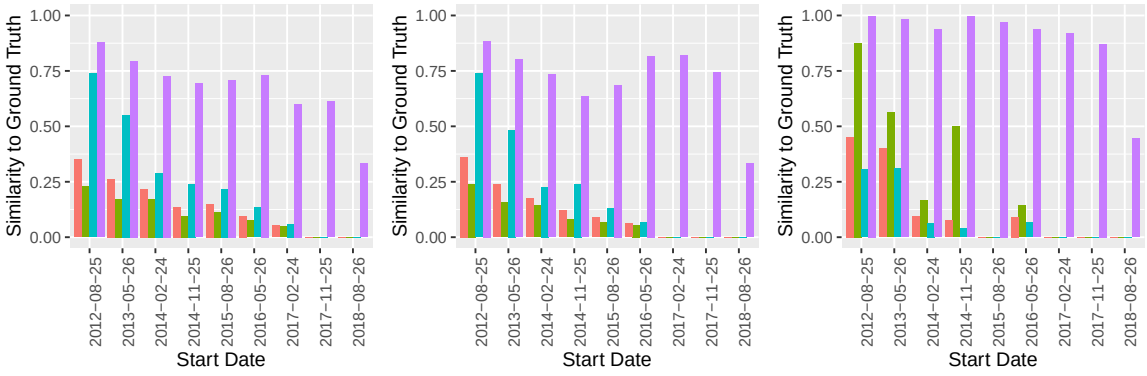
(a) Node Degree on Cochange Network (b) Eigenvector Centrality on Cochange Network (c) Hierarchy on Cochange Network



(d) Node Degree on Issue Network (e) Eigenvector Centrality on Issue Network (f) Hierarchy on Issue Network



(g) Node Degree on Cochange Issue Network (h) Eigenvector Centrality on Cochange Issue Network (i) Hierarchy on Cochange Issue Network



(j) Node Degree on Mail Network (k) Eigenvector Centrality on Mail Network (l) Eigenvector Centrality on Mail Network

Figure 6.11: ownCloud - All undirected networks that are compared to the issue-based Ground Truth. A legend can be found in Figure 6.7.

When we look at the Cochange Networks (Figure 6.11a, Figure 6.11b, Figure 6.11c), it is immediately obvious that hierarchy once again does not perform as well as the other two network classification metrics. The hierarchy classifications have a precision and specificity of consistently 100%, however the recall values are in comparison rather low as they never pass 25%. The F1 scores are low as well respectively. Both the node degree and eigenvector centrality classifications look very similar. For both, the precision and specificity values are nearly at or exactly at 100% while the recall values are on average 50% leading to an F1 score of 75%. No static increase or decrease can be seen as the values are all relatively stable.

In the Issue Networks (Figure 6.11d, Figure 6.11e, Figure 6.11f) this is slightly different. The specificity values are standing at 100% while the precision values are only between 75% and 90%. The recall values however are higher as they are 100% in the beginning but losing some value until the end where they are only at close to 50%. Important to note is the very visible drop in precision values in 2016 as this was the time when the hard fork of Nextcloud was done and in its wake many developers that worked long and hard on ownCloud left the project to work on Nextcloud. This is however only visible in the Issue Networks. This suggests that suddenly the False Positives went up. This means that the network classification metrics identify this person as relevant. This probably comes from the fact that at the beginning of this time window the *core developer*s still worked at ownCloud and triggered events which we use to identify them for our Ground Truth. However as they left, other developers that were formerly overshadowed by the work the *core developer*s did on GitHub rose up and were now identified from the network classification metric therefore leading to many FP. This drop is however not seen in the hierarchy classifications as these are very static in all values and very similar to the same plots on Cochange Networks.

When looking at Cochange Issue Networks (Figure 6.11g, Figure 6.11h, Figure 6.11i), as seen before the Issue Networks dominate the Cochange Networks. However the combination of both still does slightly better on some values especially precision wise than the Issue Networks as the Cochange Networks find some *core developer*s that were not found otherwise.

Another important note is that for this project there was a mailing list, which leads us to have *mailing list data*. When analyzing them we can however see that the mailing list was pretty much discontinued in 2017 as there is little to no data from that date on. Additionally, we can see a steady decline in all values from precision and recall to specificity.

We can therefore answer RQ1-ownCloud as if the *mailing list data* did not exist.

> *RQ1-ownCloud: Node Degree and Eigenvector Centrality on Cochange Issue Networks perform best*

As mentioned before RQ2 and RQ3 can not be answered as no committer list was found for the project.

6.2 DISCUSSION

After presenting and analyzing the results of our study in the previous section, we now discuss the three research questions in the scope of all projects and their conjunction.

6.2.1 *RQ 1*

First in Table 6.1, we remind ourselves of the analysis and the answers to our first research question:

> *RQ 1: Which of the network metrics classify core developers correctly when compared with our Ground Truth?*

Table 6.1: The answers to Research Question 1 for each individual project.

Projects	Answer to Research Question 1
Angular	Eigenvector Centrality on Cochange Issue Networks performs best.
Data Transfer Project	Node Degree and Eigenvector Centrality on Cochange Issue Networks perform best.
Keras	Hierarchy on Cochange Issue Networks performs best.
Nextcloud	Node Degree and Eigenvector Centrality on Cochange Networks perform best.
NodeJS	Eigenvector Centrality on Cochange Issue Networks performs best.
OpenSSL	Node Degree on Cochange Networks performs best.
ownCloud	Node Degree and Eigenvector Centrality on Cochange Issue Networks perform best.

What strikes most for all the different networks is that node degree and eigenvector centrality are the network classification metrics that most often perform best. Node degree performed best four times while the eigenvector centrality performed best five times. Hierarchy was only ever mentioned once. We can also see that in six out of seven cases the Cochange Issue Network had the highest values, while in the last case the Cochange Network performed best.

The node degree classification metric, the second-best method and only being slightly worse than the eigenvector centrality classification metric shows us that it catches a lot of actual *core developer*s as it not only has high precision but also high recall values. It is also interesting to note that node degree does perform slightly better when having a directed network. This suggests that the fact that the network is undirected plays an important role. The reason for this is that when we have an undirected network, any author that created for example an issue will have at least one connection to any user that works on this issue and additionally so does every other person that worked on it. This means that the difference between a *core developer* and someone just leaving a comment may only be one or two edges. However this seems to be mitigated often as the *core developer*s are working on many issues at the same time which means they are still in the top 20% of activity. When looking at the directed network this means that temporal order is respected and that any person that comments gets an edge to the developers that already worked on it. The reverse is not true. Our node degree values are so good on this network because the *core developer*s are the ones

that write the most comments on most issues because they discuss ideas and views on the project with anyone. This leads to a clearer picture of who is a *core developer* and who is not.

The eigenvector centrality metric is our best network classification metric even if only by a small margin. This shows that it can find some more *core developer*s as its network analysis is working on the context and the immediate neighbors and the influence these nodes have in the network. These better values are likely to be due to the fact that only when one node is in contact with many other nodes that are also active, i. e., through discussion with other developers, does this node also get a higher value. This also means that many ambiguities, on which developers are actually important, can be smoothed out by computing the eigenvector centrality. For the directed networks it is not easily decided which of the two networks is better for eigenvector centrality, however we can see that there is a difference between both, as we, on the one hand, compute the centrality for the undirected network taking more edges and nodes into account while, on the other hand, only using the out-going edges of a node. This last metric variation of the eigenvector centrality classification metric, i. e., taking only the out-going edges for the computation, can also be called an importance measure (indicating how many connections there are to others).

The hierarchy measure however does not fair too well in most scenarios. It is only better than the other two on Keras. For us, Keras is a special case as it is the most active out of all projects we analyzed. This means that while normally there are about 50 to 200 people working and commenting on one project in the timeline of 9 months, in Keras this is regularly multiplied by 10 or more, as we have more than 2000 users that used the issues or pull requests on GitHub. This is highly unusual considering the other projects, probably even any other OSS project. This also explains the bad values of both node degree and eigenvector centrality. On all projects, our Ground Truth finds between 0 and 50 people for each time window. This is a value that is sensible for the number of *core developer*s on an OSS project. As the network classification metrics then identify a similar number of developers as *core developer*s, we get high values for both precision and recall. For Keras this is different as our network classification metrics identify up to 600 developers as *core developer*s when using node degree and eigenvector centrality. But it only finds four *core developer*s using the hierarchy classification. As the issue-based Ground Truth also only includes four people, it is obvious that precision can not have good values, the disparity between the two is simply too big.

Concerning which networks actually produce meaningful results, we find that the combination of both the Cochange Network and the Issue Network works best, as we can analyze both the data from GitHub *issue data* as well as all commits and code artifacts. As mentioned previously, the Issue Network most of the time dominates the creation of the Cochange Issue Network as it provides the most information and resources. The Cochange Network adds very different information and hence adds some name to the list of the *core developer*s which were not on it earlier.

Additionally to finding the best network metric, both the node degree and eigenvector centrality classification metrics provide further information on big changes in the developer team.

An example of this is the clear bump in ownCloud when many *core developer*s left the team to work on Nextcloud. From this, we can see that those people were very important, but that the remaining developers in ownCloud could mitigate the sudden changes and keep the project and the developers working.

In conclusion, we can see that both the node degree and the eigenvector centrality classification metric are performing well, while the latter is slightly better. A drawback is that the classification for unusually highly active projects finds way too many *core developer*s in comparison to the real ones.

6.2.2 *RQ 2*

In Table 6.2, we remind ourselves of the research questions and its answers.

> *RQ 2: Which of the network metrics classify core developers correctly when compared to an official core developer list?*

Table 6.2: The answers to Research Question 2 for each individual project.

Projects	Answer to Research Question 2
Angular	Eigenvector Centrality on Cochange Issue Networks performs best.
Data Transfer Project	No official committer list was found.
Keras	Hierarchy on Cochange Issue Networks performs best.
Nextcloud	No official committer list was found.
NodeJS	Eigenvector Centrality on Cochange Issue Networks performs best.
OpenSSL	Node Degree and Eigenvector Centrality on Cochange Issue Networks perform best.
ownCloud	No official committer list was found.

As we were unable to find official committer lists for each project, we discuss only the projects that actually had official committer lists.

Out of the networks compared with the official committer lists three of them suggest the eigenvector centrality metric, while one suggests the node degree metric and one the hierarchy.

The node degree metric can only perform best once in four projects, suggesting that it is not really strong with the official committer lists. Additionally, the node degree classification metric is at the same time only as good as eigenvector centrality classification metric.

The eigenvector centrality classification metric displays relatively high values for three out of the four projects suggesting that the found *core developer*s are actually mostly correct, which coincides with our findings in RQ 1. We are also able to see the typical rise to the end with the official committer list as the Ground Truth, since the official committer list

is always a snapshot of the most recent developer activity. The official committer list is therefore more precise on the more current time windows. But since these values never start at 0% but at some value higher, we can infer that the developers that are now in the official committer list are actually some of the developers that were involved with the project from the beginning. This suggests that the turnover in these OSS projects is not that high at least not within the group of *core developer*s.

Keras is once again a special case as the best metric is hierarchy. The same argumentation as before counts here as well since the *core developer* list stays the same and only the Ground Truth against which we compare the values changes.

For the official committer list it also once again counts that the Cochange Issue Networks are the best as they provide more data and therefore more differentiation than when using only Cochange Networks or Issue Networks.

Furthermore, as we have seen earlier, we can find major disruptions within the developer community using our issue-based Ground Truth. This is not possible with the official committer list, as this mostly concerns the current *core developer*s and we are not able to see these leaving *core developer*s anymore.

Last but not least, we can conclude that our GT outperforms the GT based on the official committer list. This can be seen throughout nearly all plots, as the values with our GT were comparably higher. Additionally, our issue-based GT is built dynamically meaning that we do not have a disparity between the earlier and the later time windows as can be seen in many official committer list plots.

In conclusion, we can see that the eigenvector centrality metric is performing best and that the precision and recall values behave as expected. A drawback is that the classification for unusually highly active projects finds way too many *core developer*s in comparison to the real ones. We further conclude that our GT outperforms the official committer list on many occasions.

6.2.3 *RQ 3*

The last research question concerned itself with whether the official committer list and our Ground Truth were comparable. All the answers can be found in Table 6.3.

RQ 3: Is our Ground Truth comparable to the official core developer lists?

Overall, we can say that both of them were comparable. However they were never the same nor had they similar sizes. Our issue-based Ground Truth was in most cases smaller and we could therefore not get a good comparison other than the Overlap between the two. This Overlap was in most cases rather high, meaning that the official committer list included our Ground Truth.

The disparity between our Ground Truth and the official committer list can be explained through the difference between maintainers and *core developer*s as well as the time windows. To be eligible for our Ground Truth a developer needs to have triggered at least one event from the list of *elite privileges* 3.1. These events are however rare in compari-

Table 6.3: The answers to Research Question 3 for each individual project.

Projects	Answer to Research Question 2
Angular	Both of the Ground Truths overlap and have similar elements, although the official committer list contains more people than our Ground Truth.
Data Transfer Project	No official committer list was found.
Keras	Can not be compared, due to the official committer list being a project leader list.
Nextcloud	No official committer list was found.
NodeJS	Both of the Ground Truths overlap and have similar elements, although the official committer list contains more people than our Ground Truth.
OpenSSL	Both of the Ground Truths overlap and have similar elements, although the official committer list contains more people than our Ground Truth.
ownCloud	No official committer list was found.

son to many other events and only triggered rarely. This means that certain issue events are maybe triggered once a year. Additionally to this rarity, there are also maintainers that are responsible for not only answering questions but also for reviewing and subsequently merging the pull requests. On the other hand, some *core developer*s may never trigger such an event as they are not responsible for merging branches. Further disparity between our Ground Truth and the official committer list is due to the fact that our Ground Truth is dynamic and changes with time while the official committer list is a list of developers that is slowly created and not changed every other month. This results in a better performance of our GT in nearly all network classification metrics.

In conclusion, we can see that our Ground Truth finds less people than there are in the official committer lists. However, as our Ground Truth still outperforms the official committer lists, we conclude that the official committer lists consists of many developers that should not be considered as a *core developer*.

6.3 THREATS TO VALIDITY

After discussing all the results and the implications, we now present the threats to the validity of our study differentiated into internal and external threats.

Internal Validity

The first threat to validity is the validity of the data that we used to build our Ground Truth and the networks. For this data, there are many pitfalls as the same user can use different accounts or similar problems. CODEFACE mitigates this by using a heuristic from Oliva et al.

[21]. This heuristic is used to disambiguate e-mail addresses by grouping e-mails that are, for example, sent from users with the same names. This however also implies that if two users have the same name that they are handled as the same person in our data. Furthermore, there is also no possibility to group the same user for writing their name differently. Two examples for this are "Thomas Mueller" and "Thomas Müller" or "Dr. Matthias St. Pierre" and "Matthias St. Pierre". Both of these are the same person however they are not found to be the same person. To mitigate all this, we not only used the automatic approach to disambiguate users, but also went through the data manually to catch as many of these cases as possible.

Additionally, many GitHub projects, especially if they have risen to a certain size, use bots to do miscellaneous tasks for them, this includes automatic replies to issues as well as committing and pushing commits when they are thoroughly tested and approved. Thus, these bots would be considered to be *core developer*s when running our Ground Truth and the network metrics. But since we are only interested in human *core developer*s, we should not have them in our data. We therefore delete all issue events that were triggered by bots before we take any further looks at the data. The bots in the commits however are not removed as they normally only make up a small part of all commits but would delete a lot of data and respective edges to real *core developer*s when removed. To catch these robots we analyzed the GitHub repositories by hand and removed all of the ones we could find. Any bot that slipped through should have no big impact as it would be rarely used.

Last but not least, there is also the case that any GitHub user that deleted their account is no longer in their database and can therefore not be found on issues and commits alike. To circumvent a breakdown of the system, GitHub declares these users with the name "Deleted User" and an automatically generated e-mail address. This means that all users who are denoted as a deleted user would then be condensed into one user. Depending on the work done by the individual users, this deleted user could very well become a *core developer*. We therefore also delete this user wherever possible to circumvent them skewing the data.

Another threat to validity is the question of which issue events are considered to be in which of the three lists (Figure 3.1) and therefore relevant to our data. As there are nearly 50 events with an uninformative description of what they are doing and no information on which event can be triggered by whom, we tried to incorporate only the ones that were clear to be only usable by users with write permission or higher. But we can not guarantee that we did not misplace some.

Additionally, only using issue events to compute the Ground Truth is also a threat as this is only part of the way that the developers within a project work with each other. There could be *core developer*s that simply worked in the background and only ever committed but did not discuss with anybody. However, on the other hand, we could find people whose only job it was to maintain the project while not working on it themselves. These people would therefore show up in our Ground Truth, but for example never in the Cochange Network. Another threat to validity is the core threshold and the 80/20 principle. This principle is widely known and researched, however there is also the question of whether it applies in our case as well. Our data suggests it does. It might be the case that there might be a number that is better but it is highly likely looking at the good values that we have that this

better number is in the vicinity of the 80/20 principle.
Furthermore, the number of months we use for each time window is 9 months instead of the suggested 3 or 6 months [16]. This value is used because to smooth out some dynamics in the data and to collect relevant information, we need more time than 3 months and sometimes even 6 months. This value is currently researched at the Chair of Software Engineering and was therefore picked as the time window's length.
Moreover, as our network classification metrics suggest that mailing list data does not yield observable results when in combination with the other data sources, we lose out on potential information that could be relevant. On the other hand, most mailing lists are either discontinued or sparingly used once the project migrates to GitHub as can be seen with OpenSSL. The data lost is thus negligible.

External Validity

Besides the aformentioned points, it is likely that some communication and coordination is conducted via private means, meaning via a messenger or e-mail, for example. This data can of course not be analyzed and is hence not part of our study.
Aside from these, these findings can not be generalized without further work as only seven OSS projects were taken into account for this study.
Due to time limitations, we were not able to go into detail on the directed networks or pick more projects to make the results generally valid.

7

RELATED WORK

In this chapter, we will explain which work was already done on this or similar topics.

Some work has been done on the topic of analyzing socio-technical networks of OSS projects. Meneely et al. [17] used socio-technical networks based on file artifacts and commit data of a commercial product to predict failures in source code. For this they involve the churn information of revision control repositories to create the networks as well as the reported failures in the files. They found that using these networks they could accurately predict failures in files.
This was succeeded by Bird et al. [4] who evolved this concept of using socio-technical networks for failure prediction. They combined both contribution and dependency networks to get more information and make the prediction of the network more accurate. The contribution network is a network which denotes which developer has worked on which code artifact. The dependency network on the other hand concerns itself with dependencies between files and functions. Taking them together, the authors are able to show that their network identifies files that are likely to be error-prone and that the combination of the contribution and dependency network works better than either does alone.
Both of these papers concern themselves mainly with predicting failures of files and code artifacts, however the way in which they did so is very interesting as they use socio-technical networks to find them.

Furthermore Bird [3] is concerned with the way that "software affects and is affected" by relationships between developers in OSS projects. Using mailing lists and source code repositories, he analyzed multiple OSS projects. He observed that core developers have high importance in the networks and that there is "a strong relationship between development behavior and the level of importance that participants have in the social network." Our goal is similar in nature, however we want to run the data not only on mailing lists and commit data but rather a combination of them as well as issues from GitHub. Additionally, we explore this project the other way round as instead of having the identification of core developers as our main goal, we want to find out which metrics can actually be used in applications to find core developers and in retrospect peripheral developers.

Other researchers such as Xu et al. [26] use a more diverse classification identifying four groups of developers. As they are concerned with analyzing a group of projects in one big network using connections between developers when they work on the same project, this approach is not sensible for us. In our opinion this loses too much information especially on the social network as developers working on one project may never work together or talk to each other and should thus not get a connection. Hence we opted for a combination of commits, issues, and when possible mails to get a more fine-grained analysis for individual projects.

In their paper on developer coordination using networks, [13] examine how developer coordination changes over time in OSS projects using a network-analytical approach. Based on their empirical study using 18 OSS projects, they found that larger projects become scale-free and that developers are hierarchically structured at the beginning of a project, while transitioning to a hybrid in later stages. According to their study, core developers are structured in a hierarchical way. Peripheral developers on the other hand exist outside this hierarchy and are not organized as such. In our study, we use the network approach and the data collection of this paper, to obtain insights into the hierarchy between the core developers and to evaluate their usefulness.

In another paper, Joblin et al. [12] use similar principles as already mentioned, however, they are now focused on the classification of developer roles and how these rules are important for the "project's collaborative dynamics". They argue that the current standard of developer classification by using count-based metrics expedient and that network-based developer classification metrics should be used. In their study they find, that network-based developer classification metrics are indeed more successful in correctly classifying core developers. Our approach uses these findings and extends them by introducing further socio-technical connections via issues and pull requests. Additionally, we analyze only these network classification metric and evaluate which network classification metric classifies the code developers most accurately.

8 CONCLUDING REMARKS

In this final chapter, we conclude the findings of our study. Afterward, we mention the possible usage of it as well as the applicability for other studies.

8.1 CONCLUSION

This study aims to discern a network-based classification metric that correctly identifies developer roles in Open-Source Software projects. We build a Ground Truth based on issues and pull requests from GitHub by separating all issue events into categories that align with the official GitHub permissions. The networks for seven projects are built and analyzed using the *R* library Coronet. The results of the classification metrics of every network are compared against the issue-based Ground Truth. Based on this empirical comparison, we discussed different findings and facets of the classification and network relation.

First, we inspect which of the classification metrics used by Coronet has the highest precision and recall values and therefore the highest F1 score. The answer to this is for five out of seven projects the eigenvector centrality classification. It is followed by the node degree classification with a count of four. Hierarchy classification performs only once the best. For some projects we can make no definite statement which of node degree or eigenvector centrality is the better classification metric. The project in which hierarchy is the best classification metric is an outlier as the number of people that took part in the issues on GitHub are orders of magnitude bigger than in the other projects (~3000 compared to 200). Additionally, we identify that using the Cochange Issue Network for the network classification metrics leads to the best results as the most information is present. We conclude that there is no one clear answer to this research question, but that node degree and eigenvector centrality are generally better than hierarchy.

Second, we investigate if we get similar results when official committer lists that are provided by the project leads are compared against the results of the network classification metrics. For four of the projects an official committer list is found. The results are very distinct as three out of four times the eigenvector centrality classification scores the best. Both the node degree and hierarchy classifications perform the best once. Furthermore, we see that again the Cochange Issue Network is the best network to use the classification metrics on as it performs the best four out of four times. We conclude that the eigenvector centrality classification performs best.

Last but not least, we compare our Ground Truth to the official committer list to discover if we are finding the correct people with our computation. The results suggest that our Ground Truth finds the right people while not finding as many as are contained within

the official committer list. We therefore conclude that the official committer list not only contains the *core developer*s but also the developer group *maintainer*.

8.2 FUTURE WORK

Our work can be used as a baseline for further research as it proposes an automatic computation of a Ground Truth for the evaluation of socio-technical networks. One improvement to our technique can be the expansion of this Ground Truth creation to other data like mail and commits. For the commit data this could for example be the importance of a code artifact which is called very often under runtime and therefore crucial for the correct procedure of the software. This would catch more core developers that are not as active on GitHub and do not use the elite privileges that they have.

Additionally, to generalize the statements from this study, many more projects with differing sizes and different ages need to be analyzed to find a classification metric that we are confident of. Furthermore, as directed networks have similar results to the undirected ones, they should also be taken into account for the general statement.

Moreover, other centrality metrics like *Katz Centrality* [20, pp. 173–175], *Closeness Centrality* [20, pp. 183–186] and *Betweenness Centrality* [20, pp. 187–193] can also be implemented, as they might have slightly different results than the currently implemented ones. In addition, the results of the classification metrics could also be taken in conjunction and looked at together using the union of the sets. This will lead to a higher number of core developers, giving them the advantage that these developers are computed using different centrality metrics and have therefore slightly different requirements for a core developer. We plan to further refine the classification of developers so that we can accurately predict the actual *core developer*s.

A

APPENDIX

A.1 ISSUE EVENT TYPES

All issue event types of GitHub are explained in detail in this section.
All issue events that are considered to be in the *collaborative privileges* group are succeeded by the sentence "This event is considered slightly useful." in the appendix.
All issue events that are considered to be in the *elite privileges* group are succeeded by the sentence "This event is considered useful." in the appendix.
All issue events that are considered to be in the *common privileges* group are succeeded by the sentence "This event is discarded." in the appendix.

added_to_project

The event *added_to_project* is triggered when an issue or a pull request is added to a project board and the project board was enabled by the administrators for this project [36]. To interact with a project board the user needs to have *write* permission which can only be granted by an administrator. This implies that the author of the event is trusted by the administrators and they play a somewhat crucial role in the project [85].
This event is considered useful.

assigned

The event *assigned* is used when somebody assigns an issue to someone [37]. To assign someone to an issue the author of this event needs to have the permission of at least *triage* [86]. This highly suggests that the author is a maintainer or core developer with the task of working on the issues.
This event is considered slightly useful.

automatic_base_change_failed

The event *automatic_base_change_failed* is triggered by GitHub itself and does not come into play for us [38].
This event is discarded.

automatic_base_change_succeeded

The event *automatic_base_change_succeeded* is triggered by GitHub itself and does not come into play for us [39].
This event is discarded.

base_ref_changed

The event *base_ref_changed* is triggered when the base reference branch of a pull request is changed [40]. Anybody can open a pull request and hence anybody can change the base branch [34]. Even though on public repositories it is not possible for someone below *write* permission to create a pull request, the developer can still fork the project, create a pull request and then send it to the main repository.
This event is discarded.

closed

The event *closed* is triggered when an issue or a pull request is closed [41]. Anybody can open issues on the project and anybody can close issues that they opened themselves [86].
This event is discarded.

comment_deleted

The event *comment_deleted* is triggered when a comment is deleted by a user. Anybody can write a comment and delete their own comments [86].
This event is discarded.

commented

The issue event *commented* is triggered when a comment is created on an issue or a pull request [42]. Anybody can write a comment on an issue or a pull request.
This event is discarded.

committed

The issue event *committed* is triggered when a commit is added to a pull requests *HEAD* branch [43]. Anybody can create a pull request and commit to it.
This event is discarded.

connected

The event *connected* is triggered when an issue or a pull request was connected to an issue [44]. Anybody can create an issue and connect this issue to another issue by linking it with a keyword that can be written into the description [44]. Since anybody can edit their

own issues description, anybody can link an issue to another issue [86].
This event is discarded.

convert_to_draft

The pull request event *convert_to_draft* is triggered when a user changes the pull request into a draft to make it not committable [45]. The sources disagree on who is eligible to doing this. The "Repository Permissions" source [86] states that this is only available for people with *write* permission or higher while the link that is included as the source for this statement states that anybody who authored a pull request can do this to their own pull requests [33]. We will assume that [33] is correct.
This event is discarded.

converted_note_to_issue

The issue event *converted_note_to_issue* is triggered when a user converts a note that was added to a project board to an issue [31, 46]. Only people that are either a member of the organization or are invited to work on the project board can create and convert notes [84]. We assume that anybody who is invited is considered to be a core developer or more.
This event is considered useful.

created

The event *created* is triggered when a branch, issue or pull request is created or opened. Anybody can create new issues for a project [86].
This event is discarded.

cross_referenced

The event *cross_referenced* is triggered when an issue or a pull request is referenced from another issue or pull request. Anybody can create issues and refer to other issues [47].
This event is discarded.

demilestoned

The event *demilestoned* is triggered when an issue or a pull request are removed from a milestone [48]. Only people that have at least *triage* permissions on the project can apply milestones [86].
This event is considered slightly useful.

deployed

The event *deployed* is triggered when a branch was sent out for deployment [49]. Only Administrators are able to deploy branches [86].
This event is considered useful.

deployment_environment_changed

The event *deployment_environment_changed* is triggered when the deployment environment of a pull request is changed [50]. Only administrators are able to change the deployment environment [86].
This event is considered useful.

disconnected

The event *disconnected* is triggered when the linking of two issues or pull requests is removed [51]. Anybody can create issues and connect this issue to another issue by linking it with a keyword that can be written into the description. Since the editing of the description of an issue is open to the author of the issue, the author can also remove the linking and disconnect the two issues [86].
This event is discarded.

head_ref_deleted

The event *head_ref_deleted* is triggered when the *HEAD* branch of the pull request is deleted [52]. Anybody can open a pull request and hence anybody can change the head branch [34].
This event is discarded.

head_ref_restored

The event *head_ref_restored* is triggered when the *HEAD* branch of the pull request is restored to the last known *commit* [53]. Anybody can open a pull request and hence anybody can change the head branch [34].
This event is discarded.

labeled

The event *labeled* is triggered when a label is added to an issue or a pull request [54]. Only people with *triage* permissions or higher are able to apply labels [86].
This event is considered slightly useful.

locked

The event *locked* is triggered when an issue or a pull request is locked [55]. Conversations can only be locked by users with *write* permissions or higher [81, 86].
This event is considered useful.

mentioned

The event *mentioned* is triggered when a user is mentioned via *@mentioned* in an issue or pull request body [57]. Anybody can create issues and mention people in the comments of said issue [86].
This event is discarded.

marked_as_duplicate

The event *marked_as_duplicate* is triggered when a user with *triage* permission or higher marks an issue as a duplicate of another issue or a pull request as a duplicate of another pull request [56, 86].
This event is considered slightly useful.

merged

The event *merged* is triggered when a pull request is successfully merged [58]. Anybody with *push* permissions (*write* permissions) can complete the merge [82].
This event is useful.

milestoned

The event *milestoned* is triggered when an issue or a pull request is added to a milestone [59]. Only people that have at least *triage* permissions on the project can apply milestones [86].
This event is considered slightly useful.

moved_columns_in_project

The event *moved_columns_in_project* is triggered when an issue or a pull request was moved from one column to another [60]. This is only possible if project boards are enabled are by the administrators. To interact with a project board the user needs to have *write* permission which can only be granted by an administrator. This implies that the author of the event is trusted by the administrators and there plays a somewhat crucial role in the project [85].
This event is considered useful.

pinned

The event *pinned* is triggered when an issue is pinned [61]. Only people that have at least *write* permissions on the project can apply milestones and pin issues [83].
This event is considered useful.

ready_for_review

The event *ready_for_review* is triggered when a pull request is created and is not in draft mode [62]. Anybody can create pull requests and put them not in draft mode [86].
This event is discarded.

referenced

The event *referenced* is triggered when an issue is referenced from a commit message [63]. Anybody can create pull requests and commit to them with a reference to an issue [86].
This event is discarded.

referenced_by

The event *referenced_by* is triggered when somebody references another issue in an issue comment. Anybody can comment on an issue [86].
This event is discarded.

removed_from_project

The event *removed_from_project* is triggered when an issue or a pull request is removed from a project board and the project board was enabled by the administrators for this project [64]. To interact with a project board the user needs to have *write* permission which can only be granted by an administrator. This implies that the author of the event is trusted by the administrators and they play a somewhat crucial role in the project [85].
This event is considered useful.

renamed

The event *renamed* is triggered when the title of an issue or a pull request is changed [65]. Anybody can create issues and pull requests on a project and change the title when they are the issue or pull request author [86].
This event is discarded.

reopened

The event *reopened* is triggered when an issue or a pull request is reopened [66]. Anybody can create an issue, close it and reopen it themselves [86].
This event is discarded.

review_dismissed

The event *review_dismissed* is triggered when a pull request review is dismissed [67]. To dismiss a pull request review you need to have at least *write* permission on the repository [35]. This dismission is not done automatically when a change in the pull request occurs.
This event is considered useful.

review_requested

The event *review_requested* is triggered when a pull request review is requested [69]. Anybody can create pull requests but only users with *triage* permissions or higher are able to request pull request reviews [86, 87].
This event is considered slightly useful.

review_request_removed

The event *review_request_removed* is triggered when a pull request review request is removed [68]. Anybody can submit reviews on pull requests but only developers with *write* permission or higher can request reviews and remove them again [86].
This event is considered useful.

reviewed

The event *reviewed* is triggered by GitHub on the pull request when a pull request is reviewed, discarding whether this was a review that approved, commented or requested changes [70]. Anybody can submit reviews on pull requests [86].
This event is discarded.

subscribed

The event *subscribed* is triggered when someone subscribes to receive notifications for an issue or pull request [71]. Anybody can subscribe to any issue or pull request.
This event is discarded.

transferred

The event *transferred* is triggered when an issue is transferred to another repository [72]. Only users with *write* permissions or higher are able to transfer issues to another repository

within the same organization [86, 89].
This event is considered useful.

unassigned

The event *unassigned* is used when somebody unassigns an issue from someone [73]. To unassign someone from an issue the author of this event needs to have the permission of at least *triage* [86]. This highly suggests that the author is a maintainer or core developer with the task of working on the issues (as mentioned in Section A.1).
This event is considered slightly useful.

unlabeled

The event *labeled* is triggered when a label is removed from an issue or a pull request. Only people with *triage* permissions or higher are able to apply labels (as mentioned in Section A.1) [74, 86].
This event is considered slightly useful.

unlocked

The event *locked* is triggered when an issue or a pull request is unlocked. Conversations can only be unlocked by users with *write* permissions or higher (as mentioned in Section A.1) [75, 81, 86].
This event is considered useful.

unmarked_as_duplicate

The event *unmarked_as_duplicate* is triggered when a user with *triage* permission or higher unmarks an issue or a pull request after is was marked as a duplicate beforehand (as mentioned in Section A.1) [76, 86].
This event is considered slightly useful.

unpinned

The event *pinned* is triggered when an issue is unpinned [77]. Only people that have at least *write* permissions on the project can apply milestones and unpin issues (as mentioned in Section A.1) [86].
This event is considered useful.

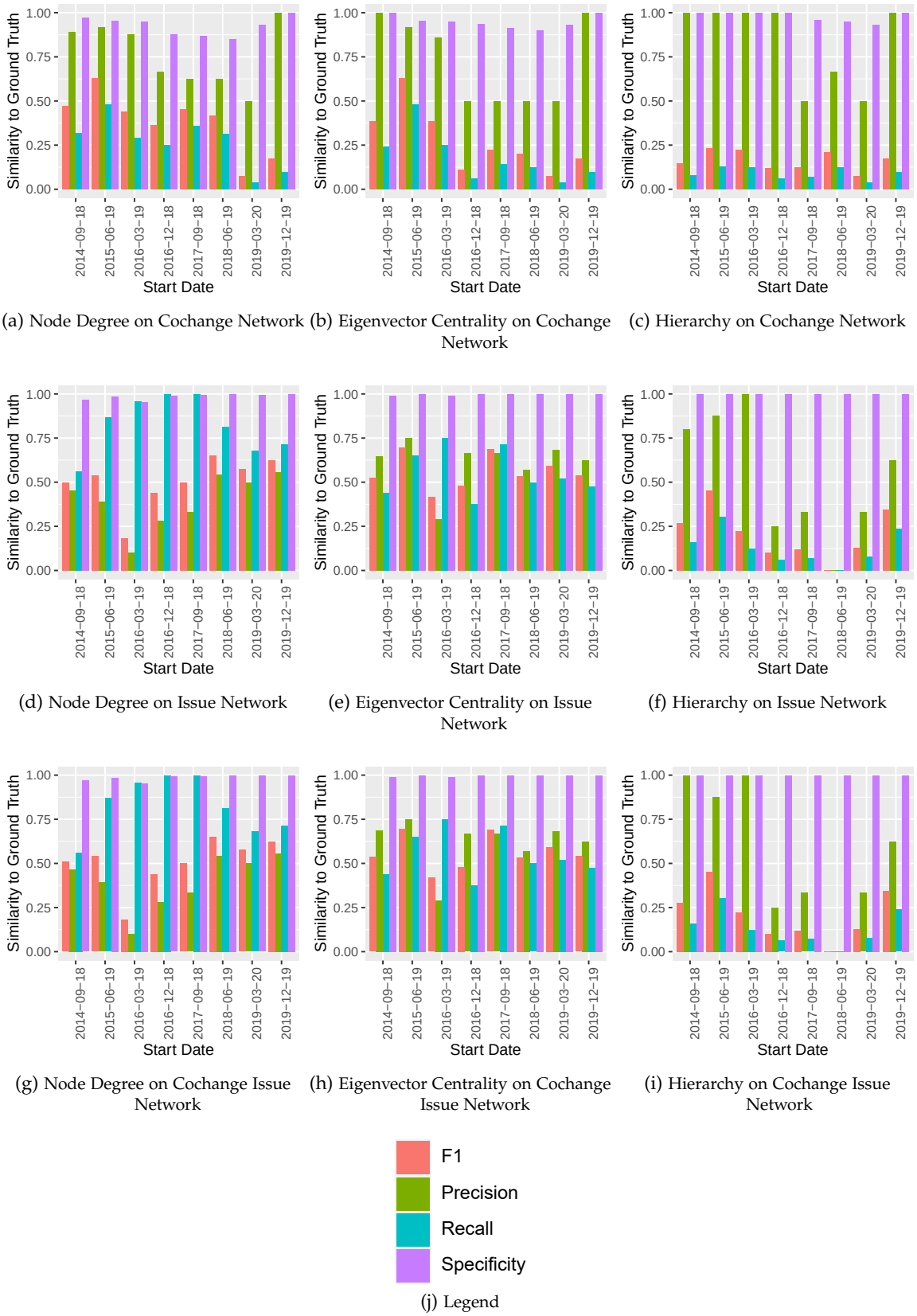
unsubscribed

The event *subscribed* is triggered when someone unsubscribes from receiving notifications for an issue or pull request [78]. Anybody can unsubscribe from any issue or pull request

they were previously subscribed to (as mentioned in Section A.1).
This event is discarded.

user_blocked

The event *user_blocked* is triggered when an organization owner blocks a user from the organization through one of the blocked user's comments on an issue [32, 79].
This event is considered useful.



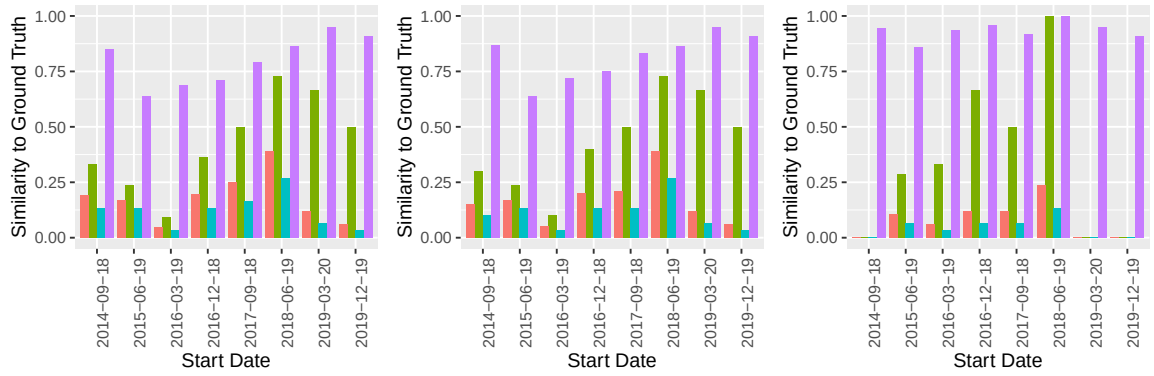
(a) Node Degree on Cochange Network (b) Eigenvector Centrality on Cochange Network (c) Hierarchy on Cochange Network

(d) Node Degree on Issue Network (e) Eigenvector Centrality on Issue Network (f) Hierarchy on Issue Network

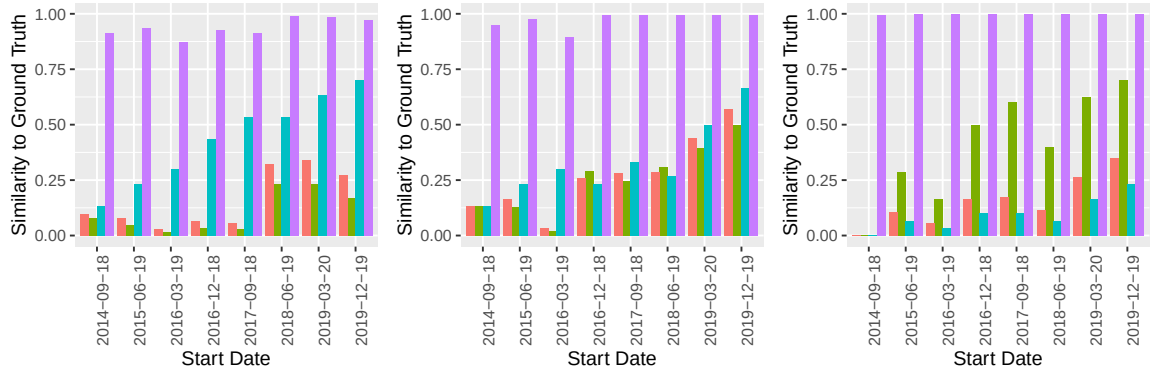
(g) Node Degree on Cochange Issue Network (h) Eigenvector Centrality on Cochange Issue Network (i) Hierarchy on Cochange Issue Network

(j) Legend

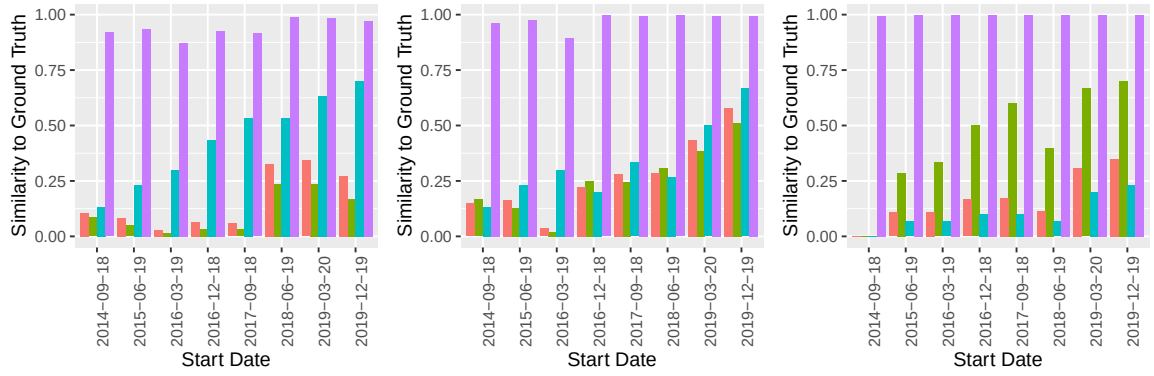
Figure A.1: Angular - All directed networks that are compared to the issue-based Ground Truth.



(a) Node Degree on Cochange Network (b) Eigenvector Centrality on Cochange Network (c) Hierarchy on Cochange Network



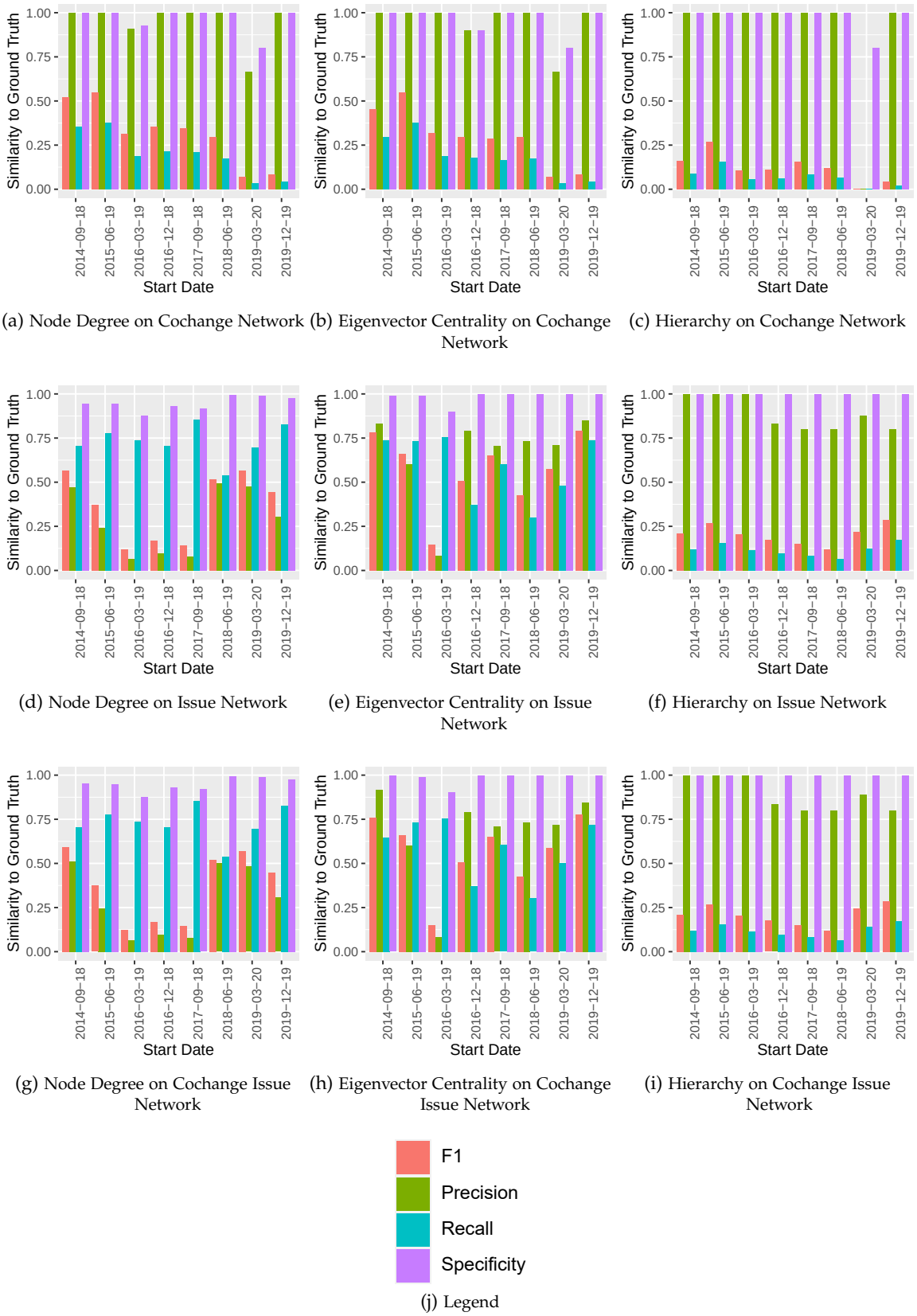
(d) Node Degree on Issue Network (e) Eigenvector Centrality on Issue Network (f) Hierarchy on Issue Network



(g) Node Degree on Cochange Issue Network (h) Eigenvector Centrality on Cochange Issue Network (i) Hierarchy on Cochange Issue Network

(j) Legend

Figure A.2: Angular - All directed networks that are compared to the issue-based Ground Truth.



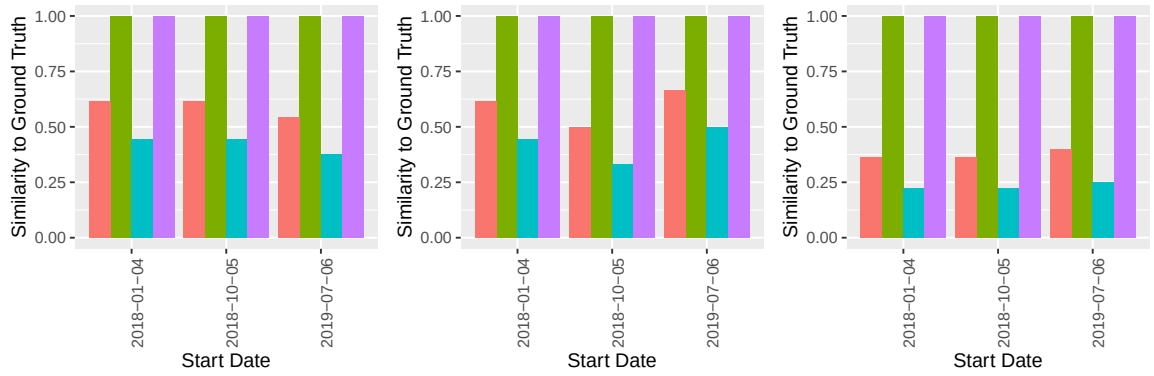
(a) Node Degree on Cochange Network (b) Eigenvector Centrality on Cochange Network (c) Hierarchy on Cochange Network

(d) Node Degree on Issue Network (e) Eigenvector Centrality on Issue Network (f) Hierarchy on Issue Network

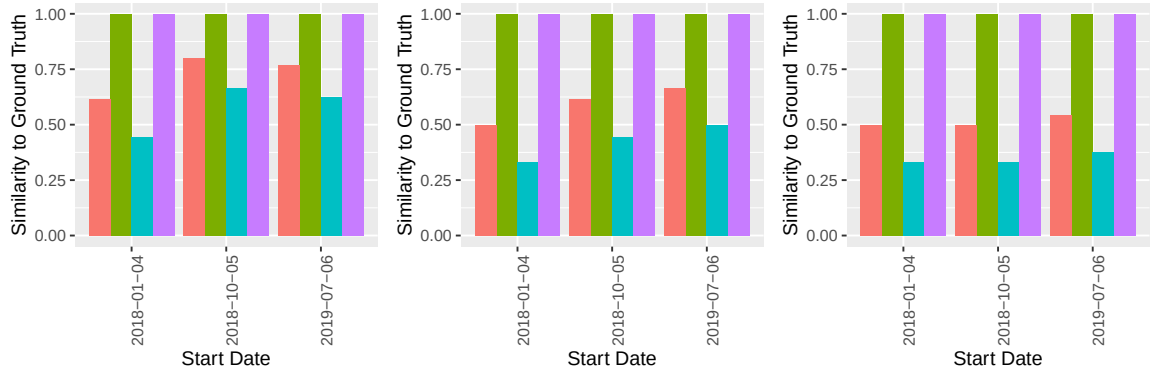
(g) Node Degree on Cochange Issue Network (h) Eigenvector Centrality on Cochange Issue Network (i) Hierarchy on Cochange Issue Network

(j) Legend

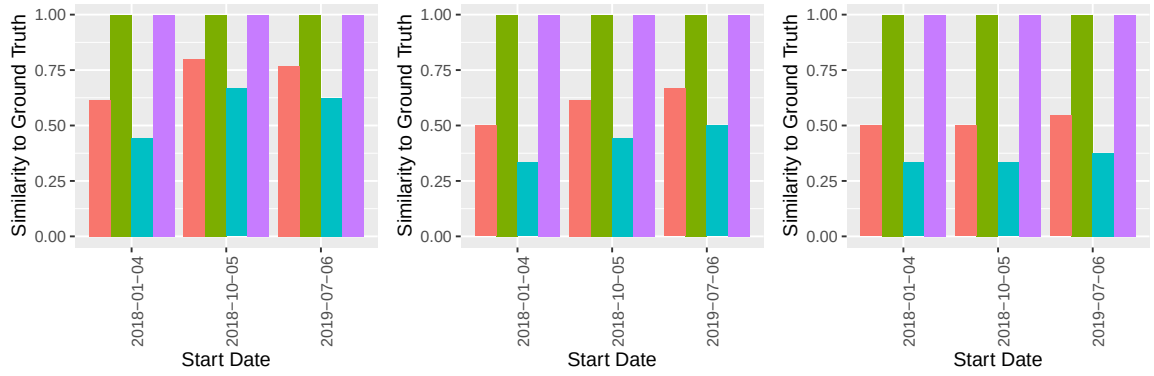
Figure A.3: Angular - All undirected networks that are compared to the issue-based Ground Truth which includes all issue events from the *collaborative privileges* list (Figure 3.1).



(a) Node Degree on Cochange Network (b) Eigenvector Centrality on Cochange Network (c) Hierarchy on Cochange Network



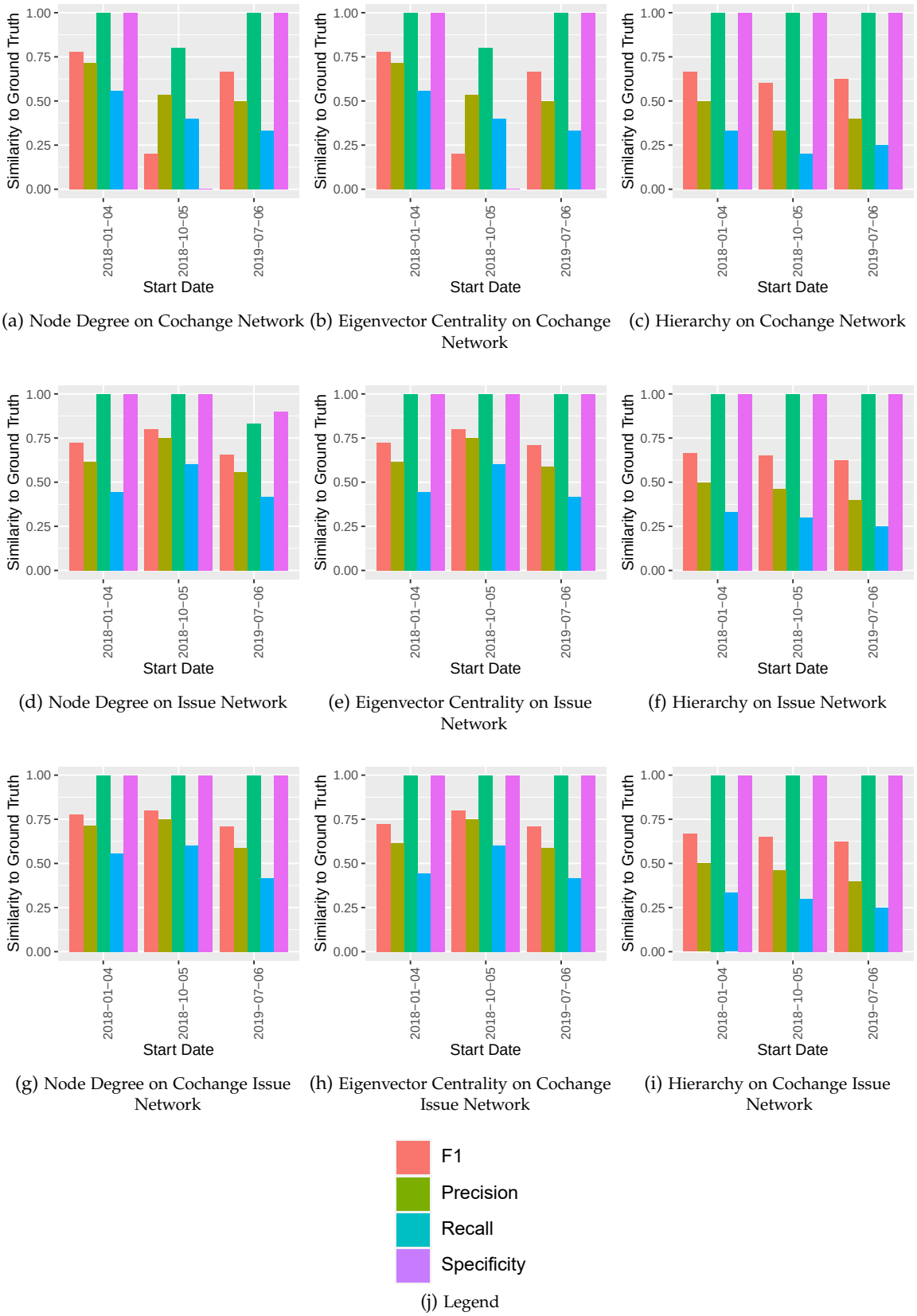
(d) Node Degree on Issue Network (e) Eigenvector Centrality on Issue Network (f) Hierarchy on Issue Network



(g) Node Degree on Cochange Issue Network (h) Eigenvector Centrality on Cochange Issue Network (i) Hierarchy on Cochange Issue Network

(j) Legend

Figure A.4: Data Transfer Project - All directed networks that are compared to the issue-based Ground Truth.



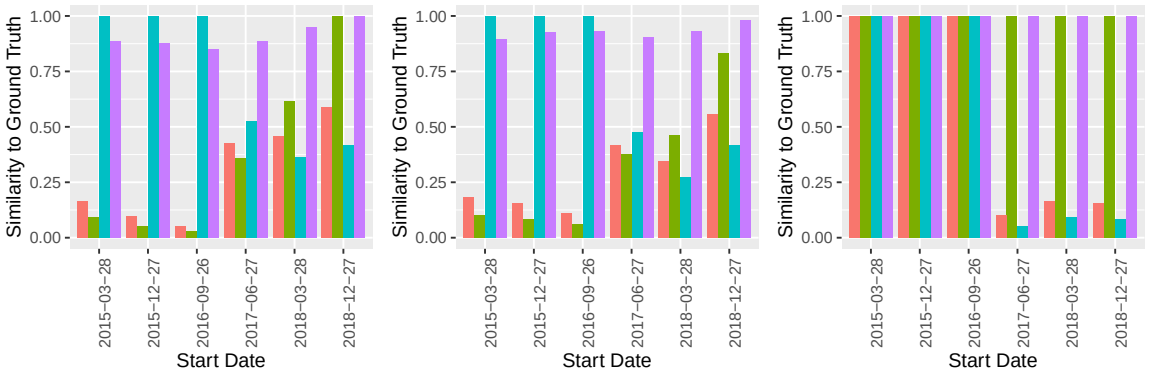
(a) Node Degree on Cochange Network (b) Eigenvector Centrality on Cochange Network (c) Hierarchy on Cochange Network

(d) Node Degree on Issue Network (e) Eigenvector Centrality on Issue Network (f) Hierarchy on Issue Network

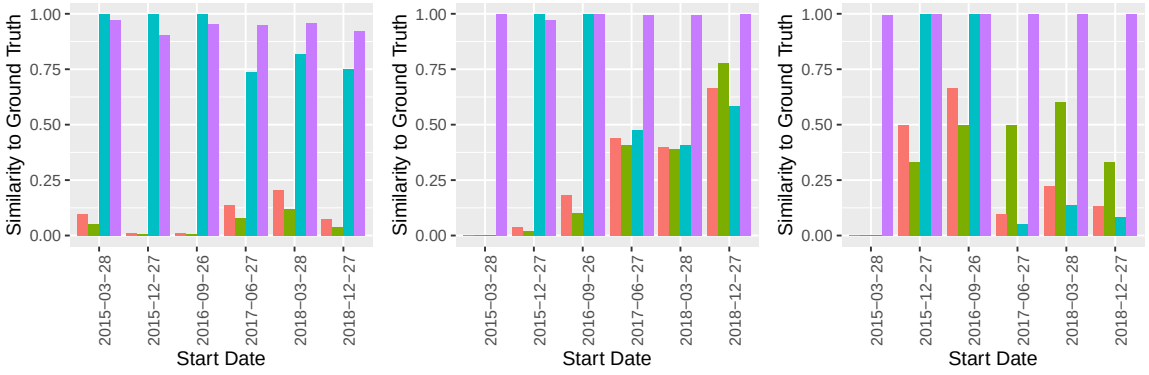
(g) Node Degree on Cochange Issue Network (h) Eigenvector Centrality on Cochange Issue Network (i) Hierarchy on Cochange Issue Network

(j) Legend

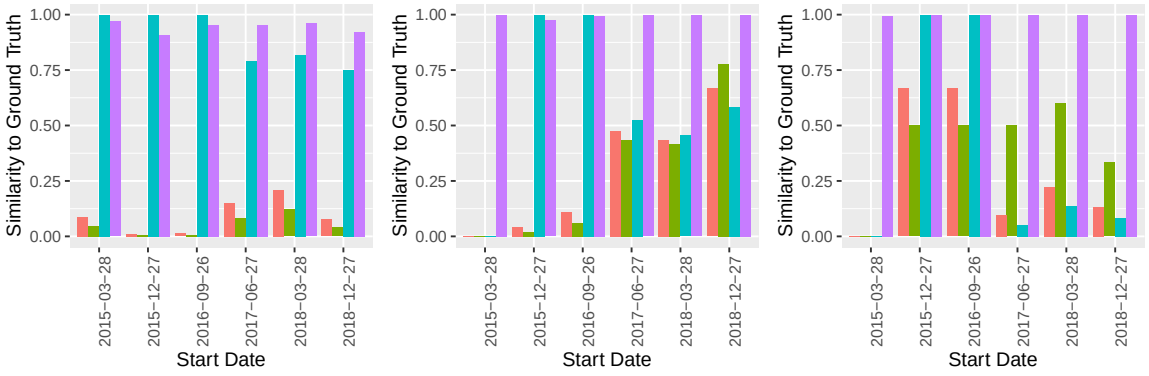
Figure A.5: Data Transfer Project - All undirected networks that are compared to the issue-based Ground Truth which includes all issue events from the *collaborative privileges* list (Figure 3.1).



(a) Node Degree on Cochange Network (b) Eigenvector Centrality on Cochange Network (c) Hierarchy on Cochange Network



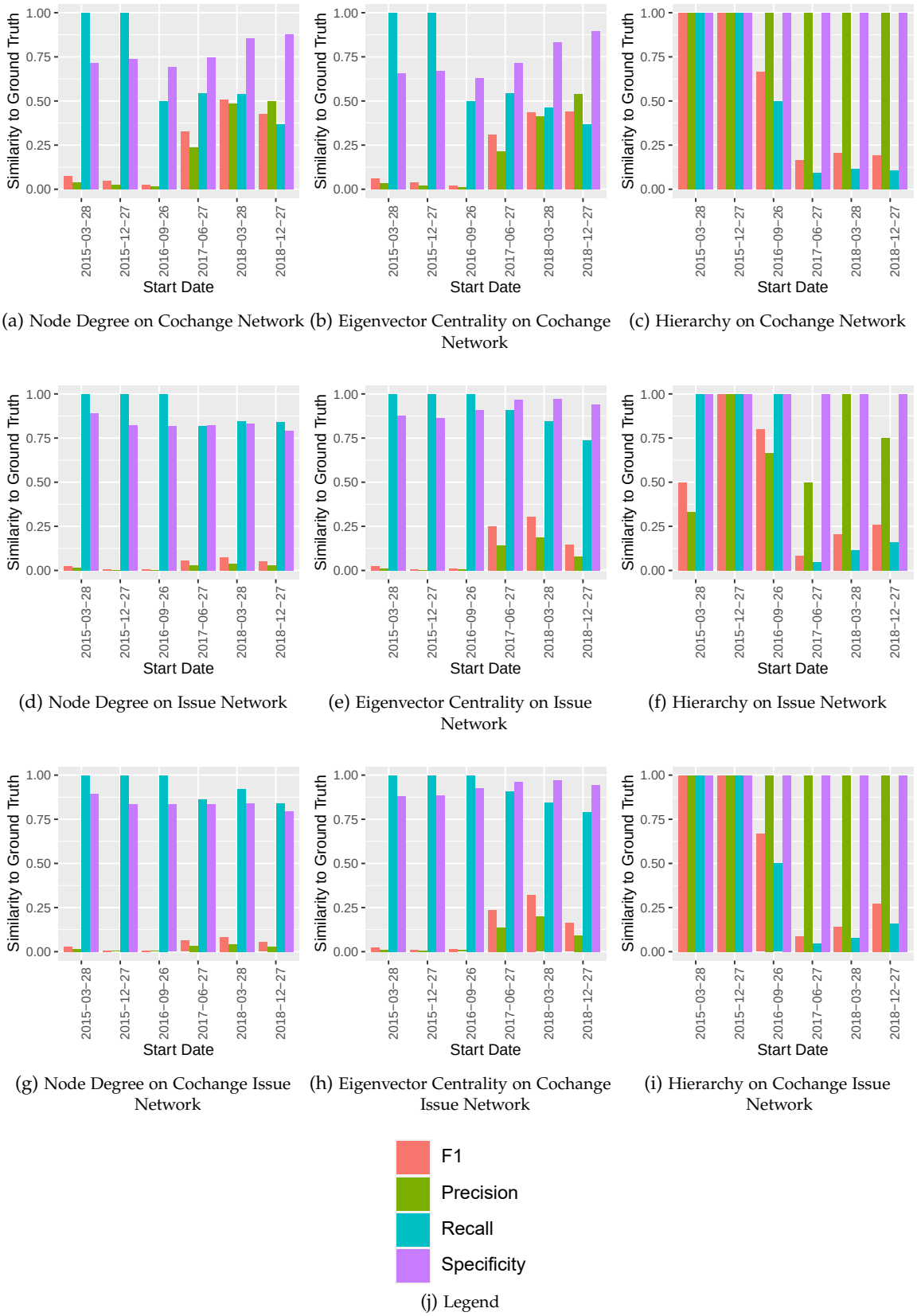
(d) Node Degree on Issue Network (e) Eigenvector Centrality on Issue Network (f) Hierarchy on Issue Network



(g) Node Degree on Cochange Issue Network (h) Eigenvector Centrality on Cochange Issue Network (i) Hierarchy on Cochange Issue Network

(j) Legend

Figure A.6: Keras - All directed networks that are compared to the issue-based Ground Truth.



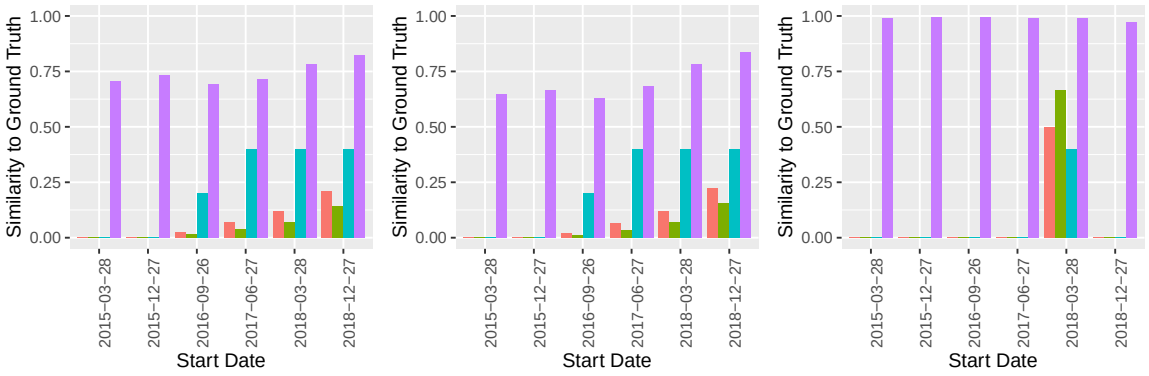
(a) Node Degree on Cochange Network (b) Eigenvector Centrality on Cochange Network (c) Hierarchy on Cochange Network

(d) Node Degree on Issue Network (e) Eigenvector Centrality on Issue Network (f) Hierarchy on Issue Network

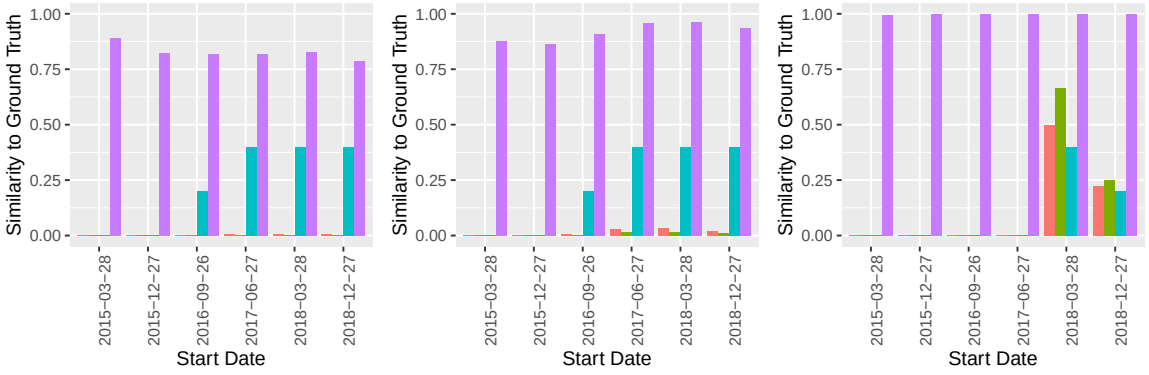
(g) Node Degree on Cochange Issue Network (h) Eigenvector Centrality on Cochange Issue Network (i) Hierarchy on Cochange Issue Network

(j) Legend

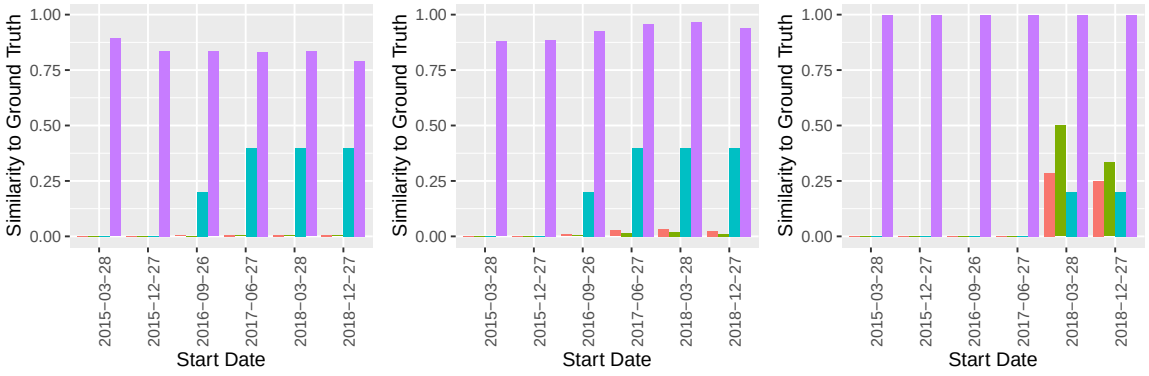
Figure A.7: Keras - All undirected networks that are compared to the issue-based Ground Truth which includes all issue events from the *collaborative privileges* list (Figure 3.1).



(a) Node Degree on Cochange Network (b) Eigenvector Centrality on Cochange Network (c) Hierarchy on Cochange Network



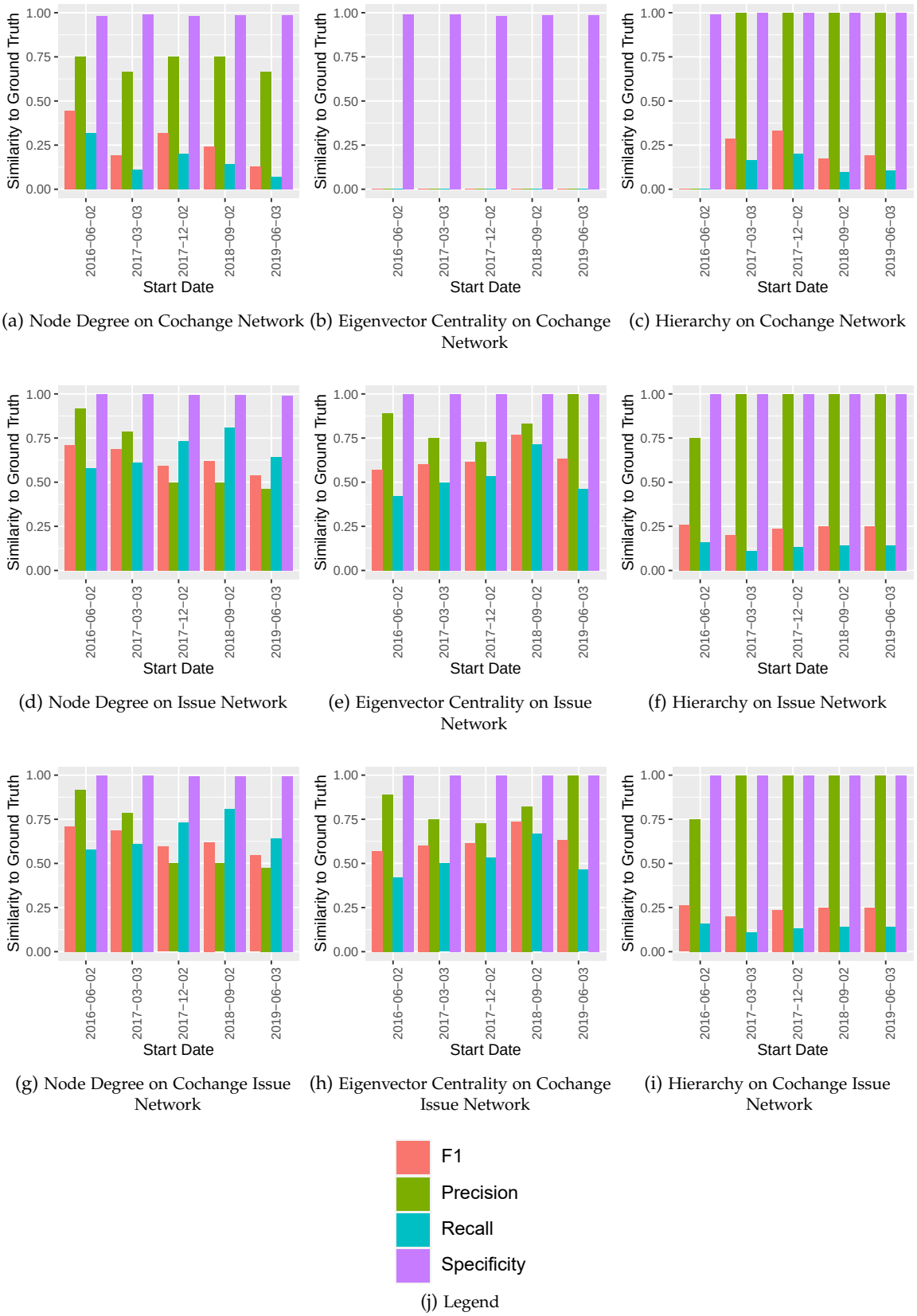
(d) Node Degree on Issue Network (e) Eigenvector Centrality on Issue Network (f) Hierarchy on Issue Network



(g) Node Degree on Cochange Issue Network (h) Eigenvector Centrality on Cochange Issue Network (i) Hierarchy on Cochange Issue Network

(j) Legend

Figure A.8: Keras - All directed networks that are compared to the issue-based Ground Truth.



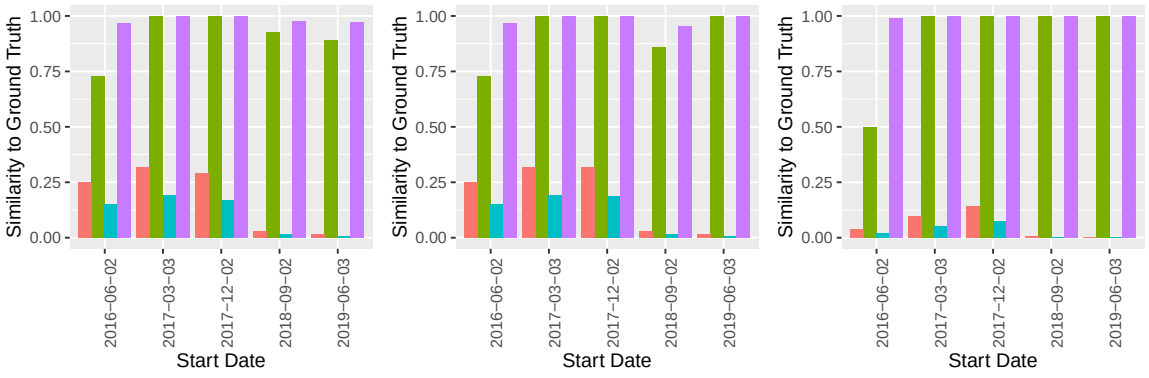
(a) Node Degree on Cochange Network (b) Eigenvector Centrality on Cochange Network (c) Hierarchy on Cochange Network

(d) Node Degree on Issue Network (e) Eigenvector Centrality on Issue Network (f) Hierarchy on Issue Network

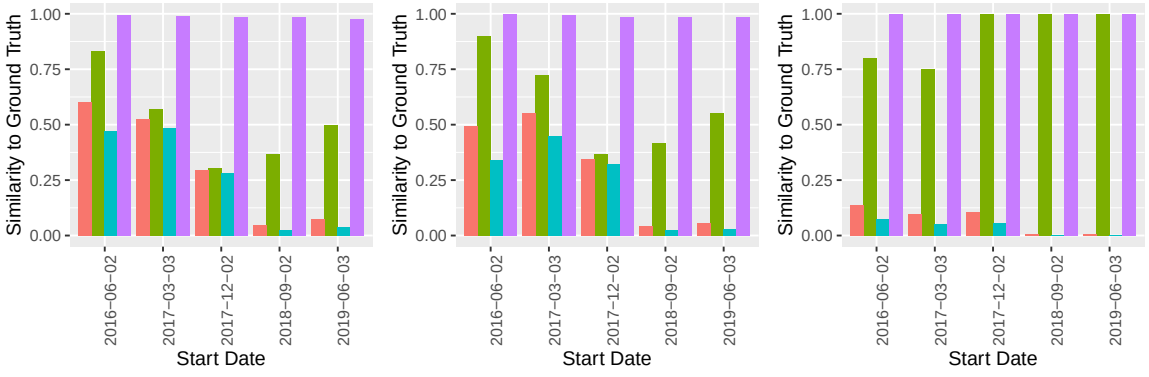
(g) Node Degree on Cochange Issue Network (h) Eigenvector Centrality on Cochange Issue Network (i) Hierarchy on Cochange Issue Network

(j) Legend

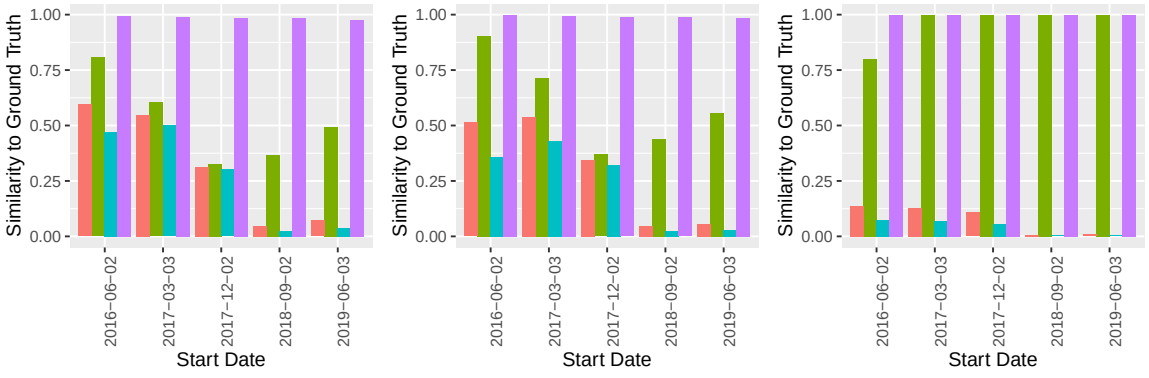
Figure A.9: Nextcloud - All directed networks that are compared to the issue-based Ground Truth.



(a) Node Degree on Cochange Network (b) Eigenvector Centrality on Cochange Network (c) Hierarchy on Cochange Network



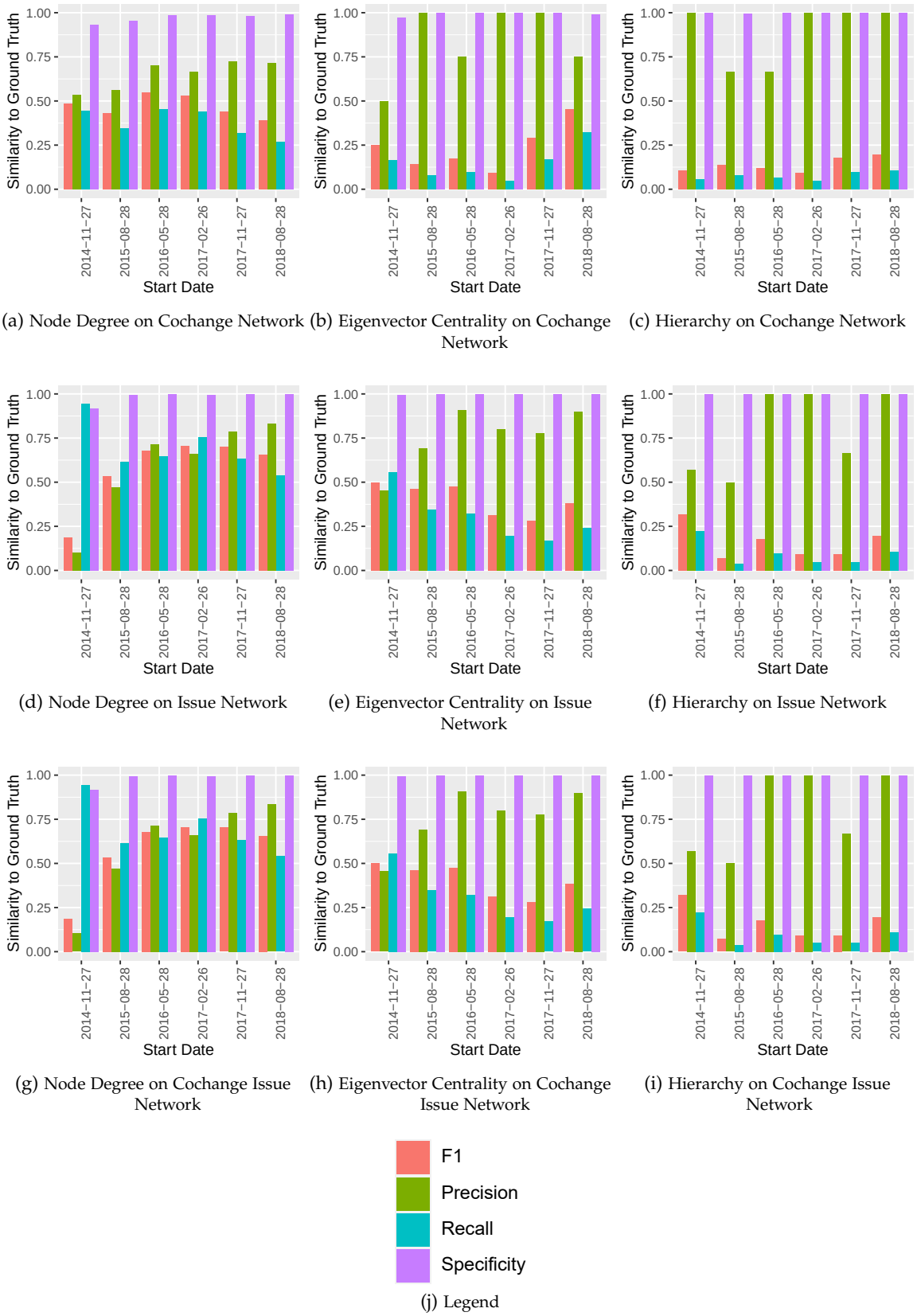
(d) Node Degree on Issue Network (e) Eigenvector Centrality on Issue Network (f) Hierarchy on Issue Network



(g) Node Degree on Cochange Issue Network (h) Eigenvector Centrality on Cochange Issue Network (i) Hierarchy on Cochange Issue Network

(j) Legend

Figure A.10: Nextcloud - All undirected networks that are compared to the issue-based Ground Truth which includes all issue events from the *collaborative privileges* list (Figure 3.1).



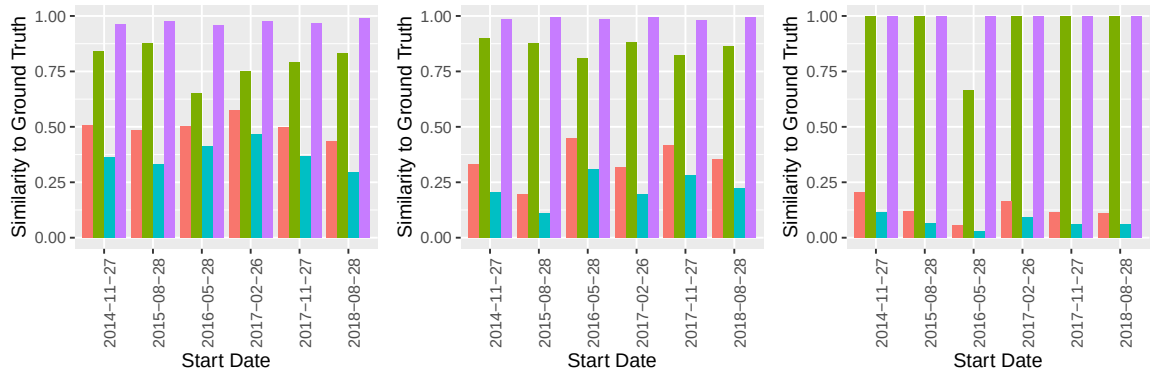
(a) Node Degree on Cochange Network (b) Eigenvector Centrality on Cochange Network (c) Hierarchy on Cochange Network

(d) Node Degree on Issue Network (e) Eigenvector Centrality on Issue Network (f) Hierarchy on Issue Network

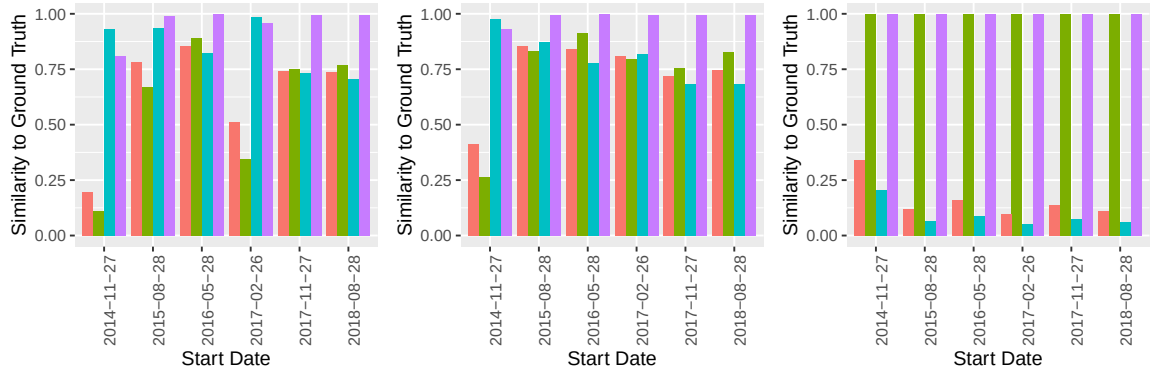
(g) Node Degree on Cochange Issue Network (h) Eigenvector Centrality on Cochange Issue Network (i) Hierarchy on Cochange Issue Network

(j) Legend

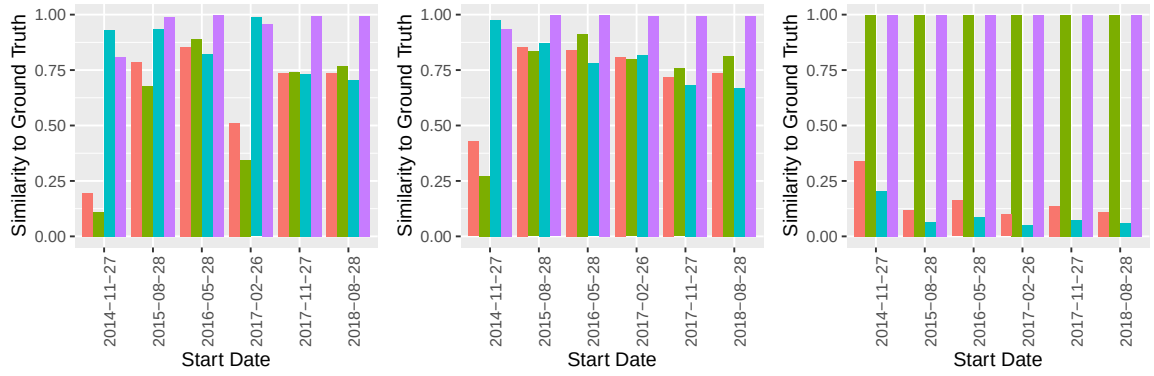
Figure A.11: NodeJS - All directed networks that are compared to the issue-based Ground Truth.



(a) Node Degree on Cochange Network (b) Eigenvector Centrality on Cochange Network (c) Hierarchy on Cochange Network



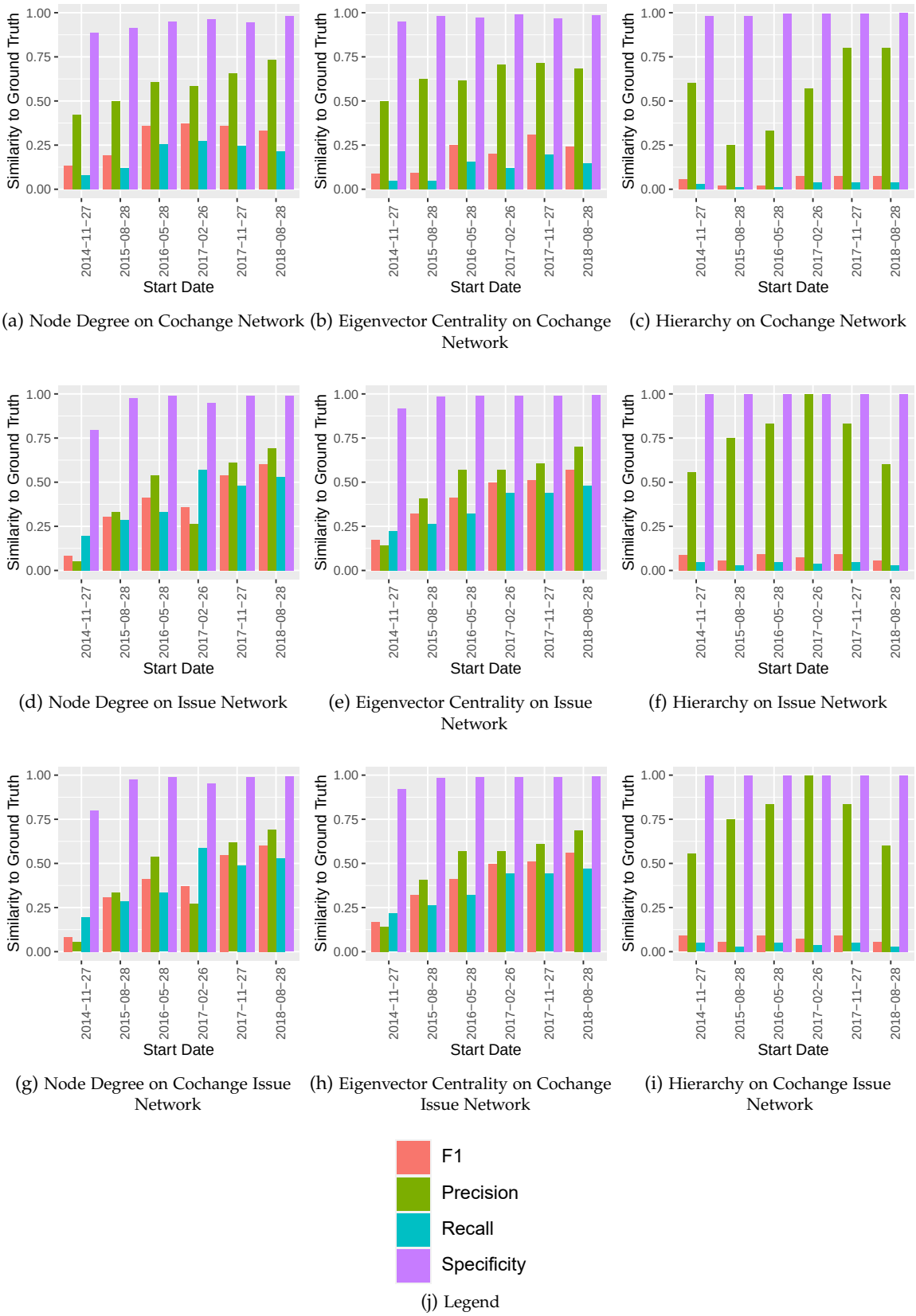
(d) Node Degree on Issue Network (e) Eigenvector Centrality on Issue Network (f) Hierarchy on Issue Network



(g) Node Degree on Cochange Issue Network (h) Eigenvector Centrality on Cochange Issue Network (i) Hierarchy on Cochange Issue Network

(j) Legend

Figure A.12: NodeJS - All undirected networks that are compared to the issue-based Ground Truth which includes all issue events from the *collaborative privileges* list (Figure 3.1).



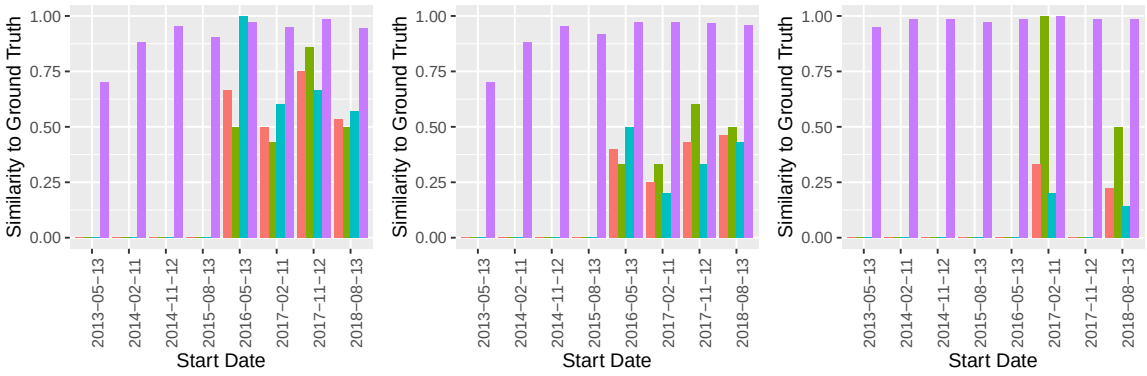
(a) Node Degree on Cochange Network (b) Eigenvector Centrality on Cochange Network (c) Hierarchy on Cochange Network

(d) Node Degree on Issue Network (e) Eigenvector Centrality on Issue Network (f) Hierarchy on Issue Network

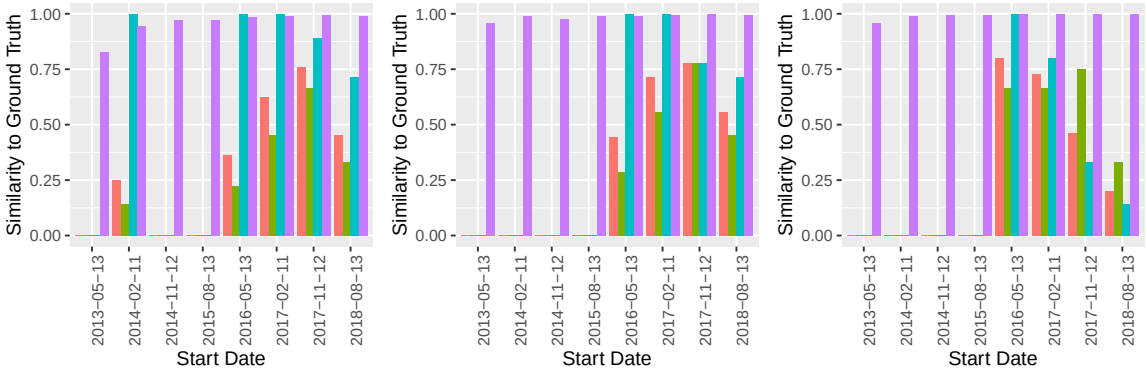
(g) Node Degree on Cochange Issue Network (h) Eigenvector Centrality on Cochange Issue Network (i) Hierarchy on Cochange Issue Network

(j) Legend

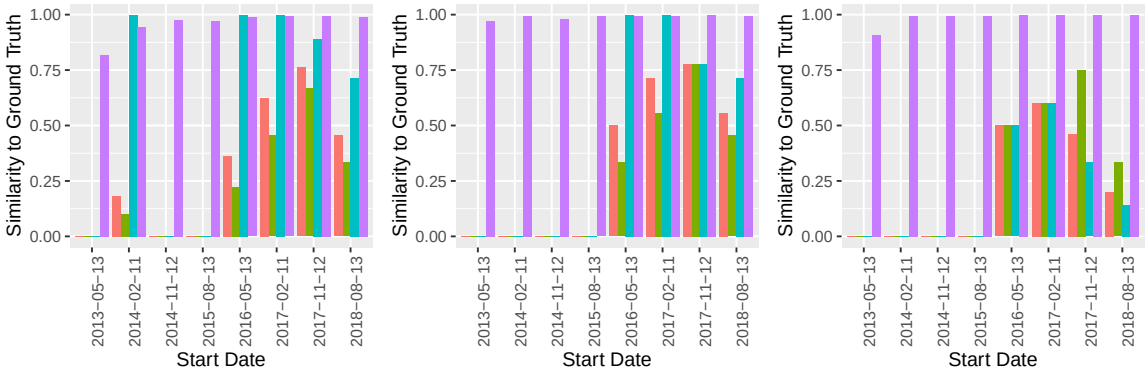
Figure A.13: NodeJS - All directed networks that are compared to the issue-based Ground Truth.



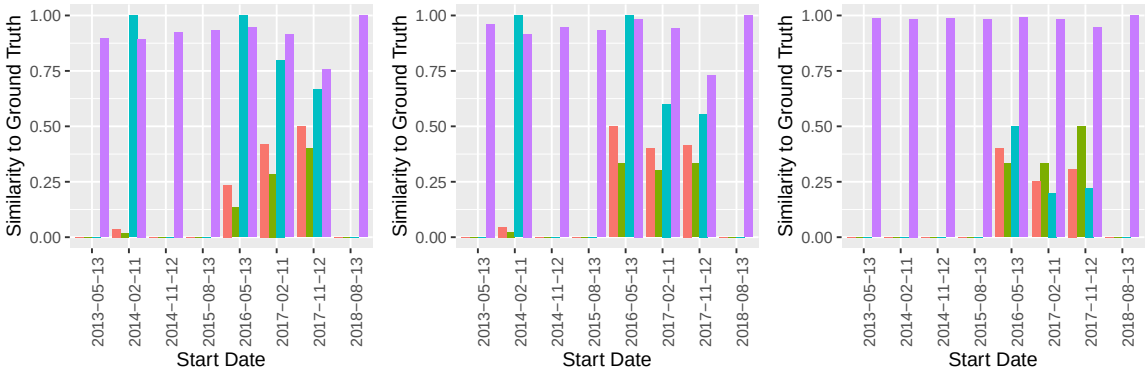
(a) Node Degree on Cochange Network (b) Eigenvector Centrality on Cochange Network (c) Hierarchy on Cochange Network



(d) Node Degree on Issue Network (e) Eigenvector Centrality on Issue Network (f) Hierarchy on Issue Network

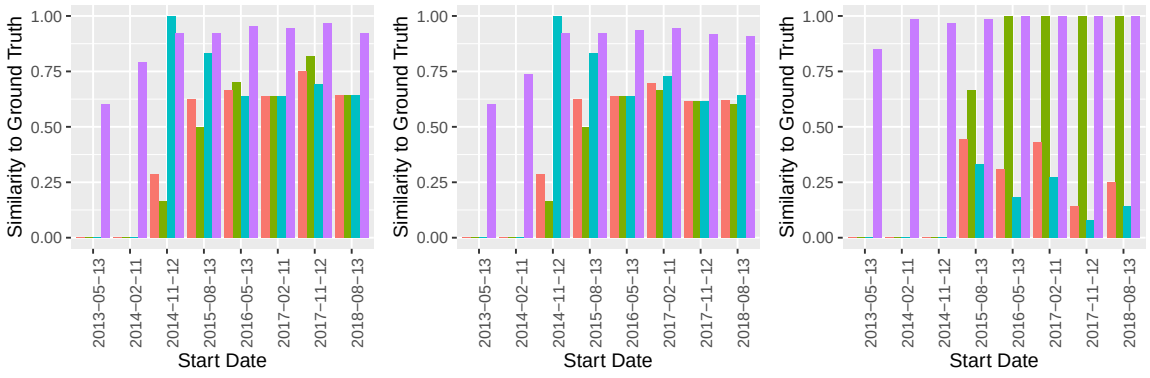


(g) Node Degree on Cochange Issue Network (h) Eigenvector Centrality on Cochange Issue Network (i) Hierarchy on Cochange Issue Network

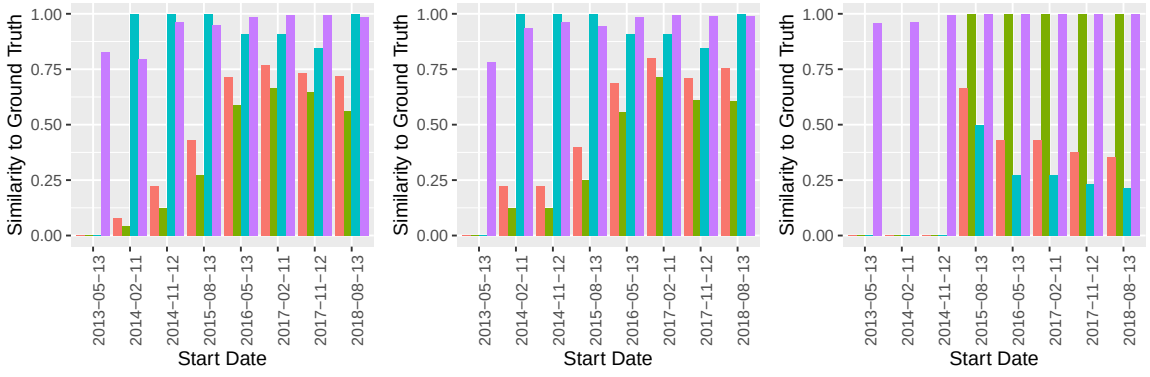


(j) Node Degree on Mail Network (k) Eigenvector Centrality on Mail Network (l) Hierarchy on Mail Network

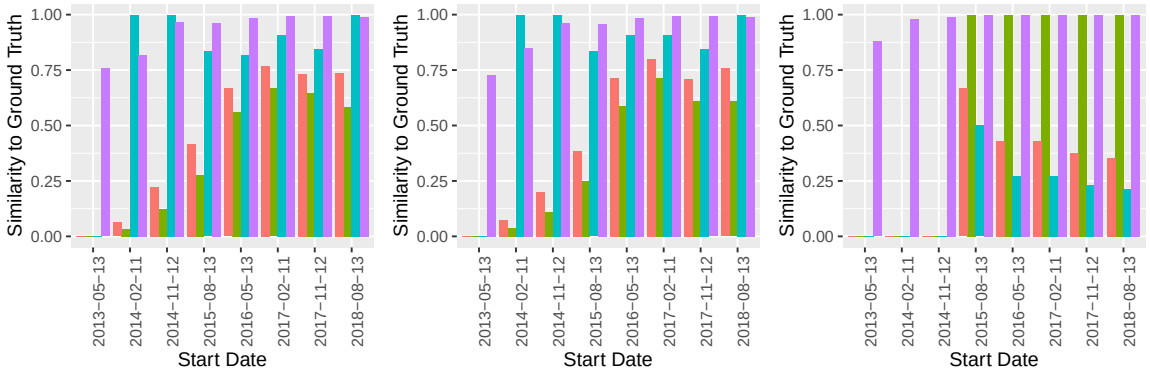
Figure A.14: OpenSSL - All directed networks that are compared to the issue-based Ground Truth. A legend can be found in Figure A.11.



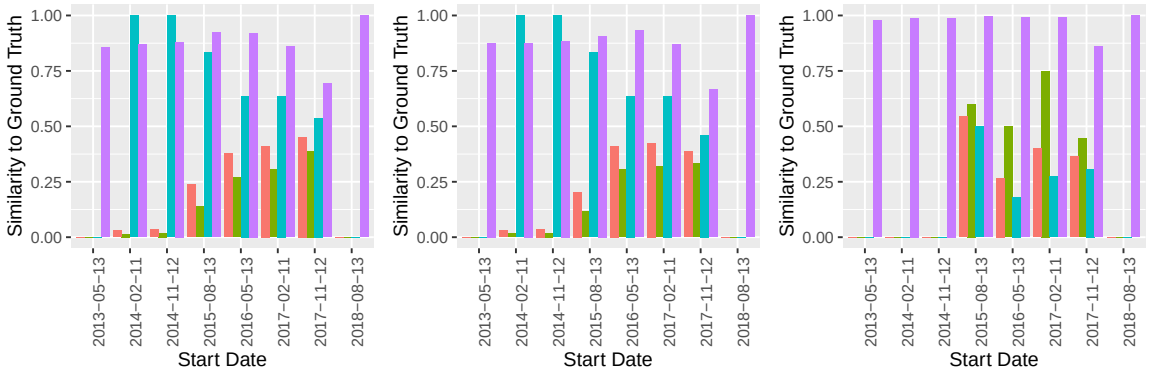
(a) Node Degree on Cochange Network (b) Eigenvector Centrality on Cochange Network (c) Hierarchy on Cochange Network



(d) Node Degree on Issue Network (e) Eigenvector Centrality on Issue Network (f) Hierarchy on Issue Network

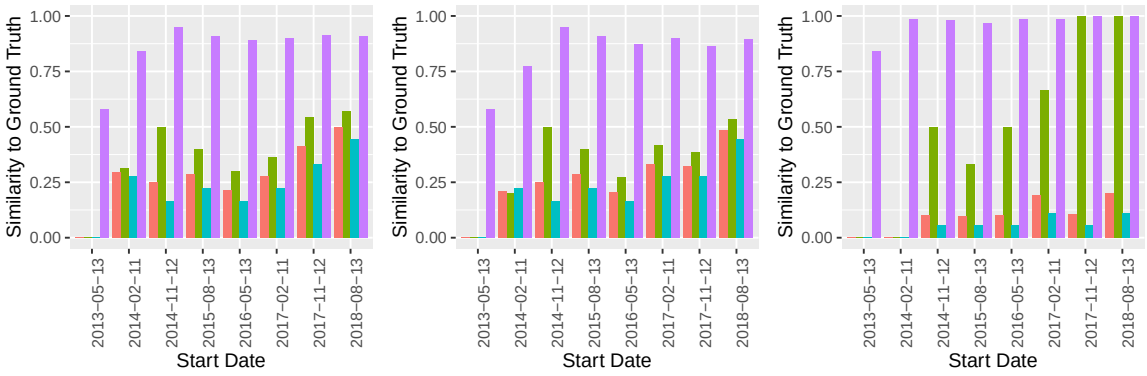


(g) Node Degree on Cochange Issue Network (h) Eigenvector Centrality on Cochange Issue Network (i) Hierarchy on Cochange Issue Network

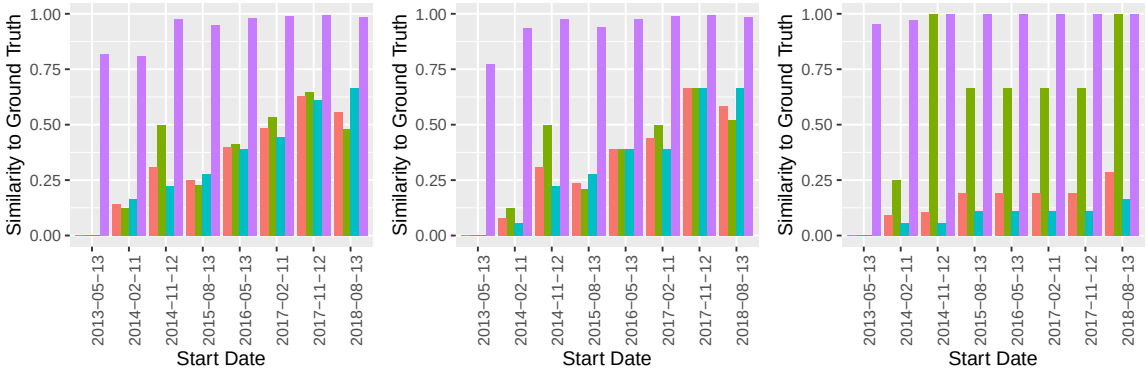


(j) Node Degree on Mail Network (k) Eigenvector Centrality on Mail Network (l) Hierarchy on Mail Network

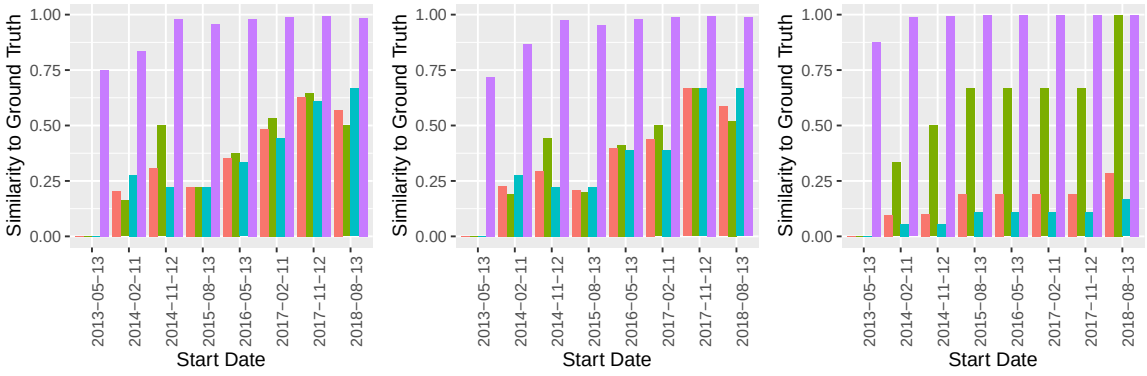
Figure A.15: OpenSSL - All undirected networks that are compared to the issue-based Ground Truth which includes all issue events from the *collaborative privileges* list (Figure 3.1). A legend can be found in Figure A.12.



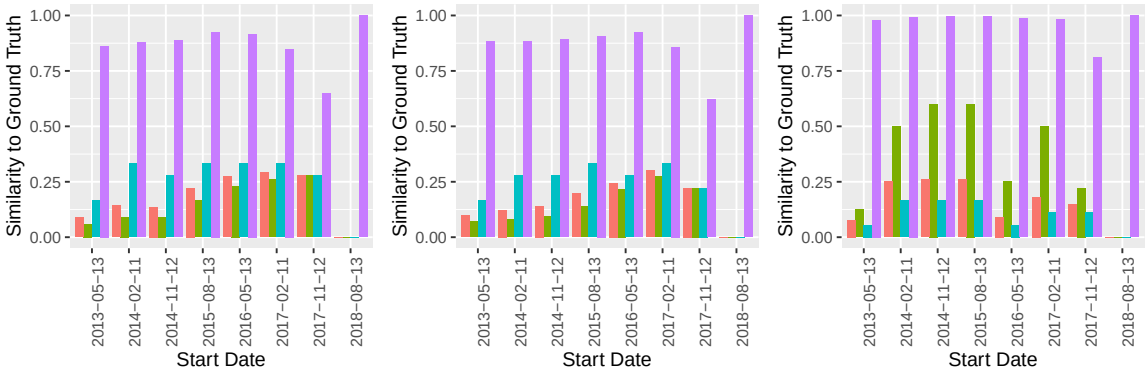
(a) Node Degree on Cochange Network (b) Eigenvector Centrality on Cochange Network (c) Hierarchy on Cochange Network



(d) Node Degree on Issue Network (e) Eigenvector Centrality on Issue Network (f) Hierarchy on Issue Network

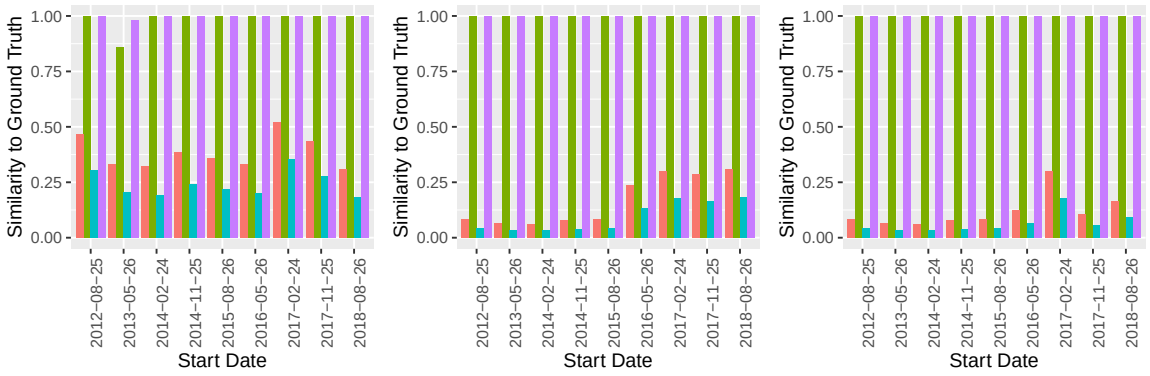


(g) Node Degree on Cochange Issue Network (h) Eigenvector Centrality on Cochange Issue Network (i) Hierarchy on Cochange Issue Network

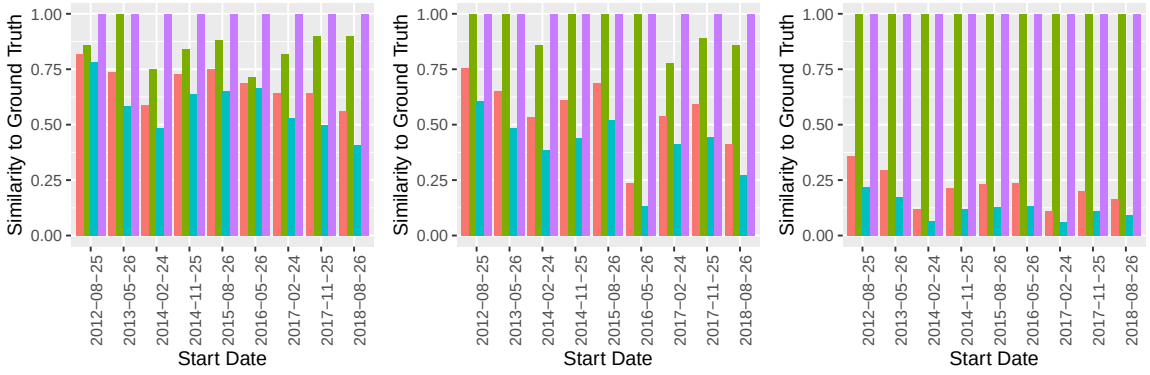


(j) Node Degree on Mail Network (k) Eigenvector Centrality on Mail Network (l) Hierarchy on Mail Network

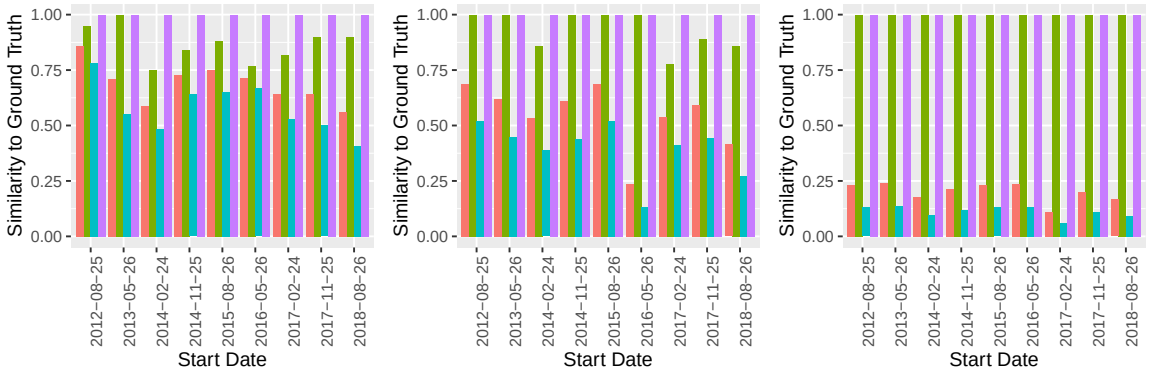
Figure A.16: OpenSSL - All directed networks that are compared to the issue-based Ground Truth. A legend can be found in Figure A.13.



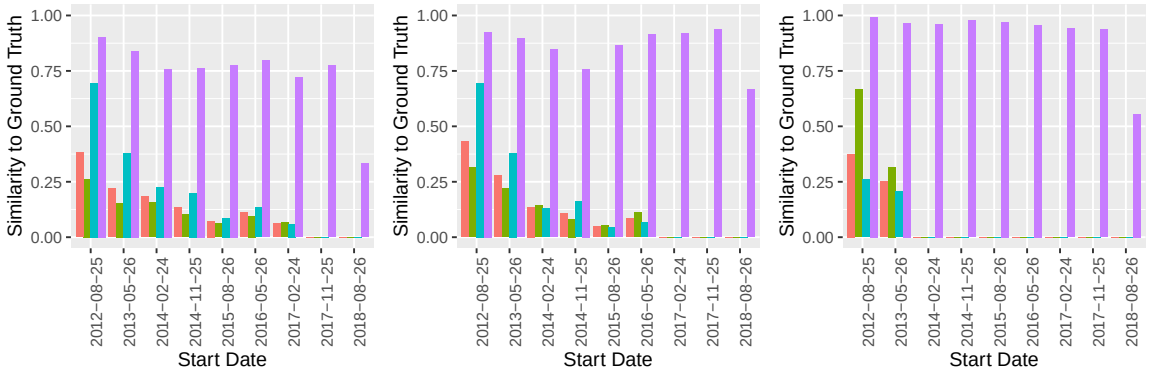
(a) Node Degree on Cochange Network (b) Eigenvector Centrality on Cochange Network (c) Hierarchy on Cochange Network



(d) Node Degree on Issue Network (e) Eigenvector Centrality on Issue Network (f) Hierarchy on Issue Network

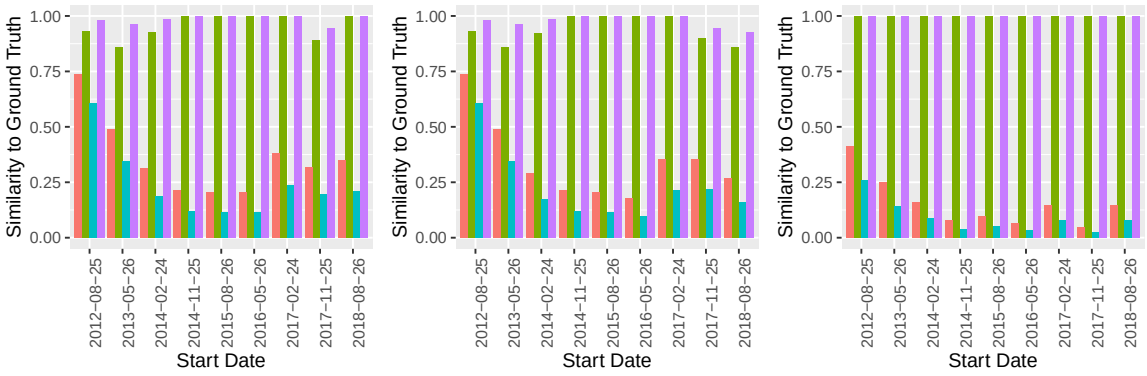


(g) Node Degree on Cochange Issue Network (h) Eigenvector Centrality on Cochange Issue Network (i) Hierarchy on Cochange Issue Network

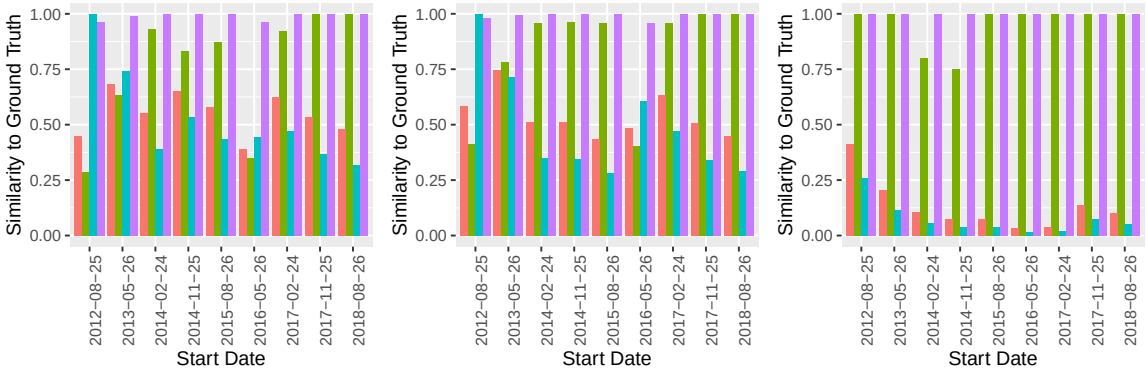


(j) Node Degree on Mail Network (k) Eigenvector Centrality on Mail Network (l) Eigenvector Centrality on Mail Network

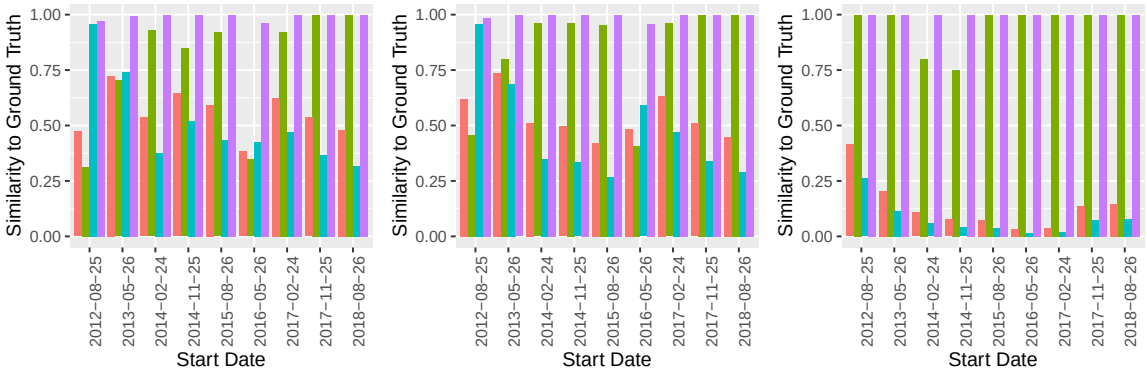
Figure A.17: ownCloud - All directed networks that are compared to the issue-based Ground Truth. A legend can be found in Figure A.11.



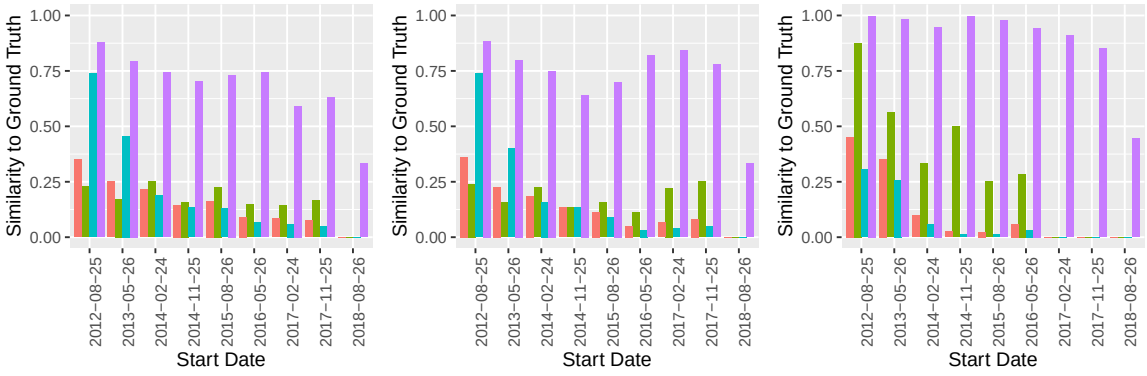
(a) Node Degree on Cochange Network (b) Eigenvector Centrality on Cochange Network (c) Hierarchy on Cochange Network



(d) Node Degree on Issue Network (e) Eigenvector Centrality on Issue Network (f) Hierarchy on Issue Network



(g) Node Degree on Cochange Issue Network (h) Eigenvector Centrality on Cochange Issue Network (i) Hierarchy on Cochange Issue Network



(j) Node Degree on Mail Network (k) Eigenvector Centrality on Mail Network (l) Eigenvector Centrality on Mail Network

Figure A.18: ownCloud - All undirected networks that are compared to the issue-based Ground Truth which includes all issue events from the *collaborative privileges* list (Figure 3.1). A legend can be found in Figure A.12.

BIBLIOGRAPHY

[1] Guilherme Avelino, Eleni Constantinou, Marco Tulio Valente, and Alexander Serebrenik. *On the abandonment and survival of open source projects: An empirical investigation*. 2019. arXiv: 1906.08058 [cs.SE] (cit. on p. 25).

[2] Bruce Avolio, Fred Walumbwa, and Todd Weber. "Leadership: Current Theories, Research, and Future Directions." In: *Annual review of psychology* 60 (Jan. 2009), pp. 421–449. DOI: 10.1146/annurev.psych.60.110707.163621 (cit. on p. 2).

[3] Christian Bird. "Sociotechnical Collaboration and Coordination in Open Source Software." In: Sept. 2011, pp. 568–573. DOI: 10.1109/ICSM.2011.6080832 (cit. on p. 65).

[4] Christian Bird, Nachiappan Nagappan, Harald Gall, Brendan Murphy, and Premkumar Devanbu. "Putting It All Together: Using Socio-technical Networks to Predict Failures." In: Nov. 2009, pp. 109–119. DOI: 10.1109/ISSRE.2009.17 (cit. on p. 65).

[5] K. Crowston, Kangning Wei, Qing Li, and J. Howison. "Core and Periphery in Free/Libre and Open Source Software Team Communications." In: *Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS'06)*. Vol. 6. 2006, 118a–118a. DOI: 10.1109/HICSS.2006.101 (cit. on p. 27).

[6] B. Dhanalaxmi, D. A. Naidu, and Dr. K. Anuradha. "A Review on Software Fault Detection and Prevention Mechanism in Software Development Activities." In: 2015 (cit. on p. 1).

[7] Reinhard Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Second. Springer, 2000 (cit. on pp. 5, 8).

[8] Stefan Dietze. "Agile requirements definition for software improvement and maintenance in open source software development." In: (Jan. 2005) (cit. on p. 25).

[9] Fabio Ferreira, Luciana Silva, and Marco Valente. "Turnover in Open-Source Projects: The Case of Core Developers." In: Oct. 2020. DOI: 10.1145/3422392.3422433 (cit. on p. 2).

[10] Matthieu Foucault, Marc Palyart, Xavier Blanc, Gail C. Murphy, and Jean-Rémy Falleri. "Impact of Developer Turnover on Quality in Open-Source Software." In: ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, pp. 829–841. ISBN: 9781450336758. DOI: 10.1145/2786805.2786870. URL: https://doi.org/10.1145/2786805.2786870 (cit. on p. 2).

[11] Paul Jaccard. "Distribution comparée de la flore alpine dans quelques régions des Alpes occidentales et orientales." In: *Bulletin de la Murithienne* 31 (1902), pp. 81–92 (cit. on p. 18).

[12] Mitchell Joblin, Sven Apel, Claus Hunsen, and Wolfgang Mauerer. "Classifying Developers into Core and Peripheral: An Empirical Study on Count and Network Metrics." In: *CoRR* abs/1604.00830 (2016). arXiv: 1604.00830. URL: http://arxiv.org/abs/1604.00830 (cit. on pp. 2, 24, 66).

[13] Mitchell Joblin, Sven Apel, and Wolfgang Mauerer. "Evolutionary Trends of Developer Coordination: A Network Approach." In: *CoRR* abs/1510.06988 (2015). arXiv: 1510.06988. URL: http://arxiv.org/abs/1510.06988 (cit. on pp. 2, 24, 25, 66).

[14] Ljubomir Lazic. "Software Errors Analysis and Prevention." In: (Mar. 2006) (cit. on p. 1).

[15] Ferdinand Malcher, Johannes Hoppe, and Danny Koppenhagen. *Angular - Grundlagen, fortgeschrittene Themen und Best Practices*. Third. Available online at https://angular-buch.com/assets/angular-buch.com_leseprobe_3.auflage.pdf; visited on October 28th, 2020. dpunkt.verlag, 2020. ISBN: 978-3-86490-779-1 (cit. on p. 32).

[16] A. Meneely and L. Williams. "Socio-technical developer networks: should we trust our measurements?" In: *2011 33rd International Conference on Software Engineering (ICSE)*. 2011, pp. 281–290. DOI: 10.1145/1985793.1985832 (cit. on pp. 27, 63).

[17] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. "Predicting failures with developer networks and social network analysis." In: Jan. 2008, pp. 13–23. DOI: 10.1145/1453101.1453106 (cit. on p. 65).

[18] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. "Two Case Studies of Open Source Software Development: Apache and Mozilla." In: *ACM Trans. Softw. Eng. Methodol.* 3 (July 2002), pp. 309–346. ISSN: 1049-331X. DOI: 10.1145/567793.567795. URL: https://doi.org/10.1145/567793.567795 (cit. on p. 27).

[19] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki NISHINAKA, Kouichi Kishida, and Yunwen Ye. "Evolution Patterns of Open-Source Software Systems and Communities." In: *International Workshop on Principles of Software Evolution (IWPSE)* (Jan. 2003). DOI: 10.1145/512035.512055 (cit. on p. 24).

[20] Mark Newman. *Networks: An Introduction*. USA: Oxford University Press, Inc., 2010. ISBN: 0199206651 (cit. on pp. 9, 68).

[21] Gustavo Oliva, Francisco Santana, Kleverton Oliveira, Cleidson Souza, and Marco Aurelio Gerosa. "Characterizing Key Developers: A Case Study with Apache Ant." In: vol. 7493. Sept. 2012, pp. 97–112. DOI: 10.1007/978-3-642-33284-5_8 (cit. on p. 61).

[22] Vilfredo Pareto. *Manuale di Economia Politica*. (English translation, A. M. Kelly, (1971)). 1909 (cit. on p. 27).

[23] David Powers. "Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness & Correlation." In: *Mach. Learn. Technol.* 2 (Jan. 2008) (cit. on p. 19).

[24] Erzsébet Ravasz and Albert-László Barabási. "Hierarchical organization in complex networks." In: *Physical Review E* 67.2 (2003). ISSN: 1095-3787. DOI: 10.1103/physreve.67.026112. URL: http://dx.doi.org/10.1103/PhysRevE.67.026112 (cit. on p. 10).

[25] MK Vijaymeena and K Kavitha. "A survey on similarity measures in text mining." In: *Machine Learning and Applications: An International Journal* 3.2 (2016), pp. 19–28 (cit. on p. 19).

[26] J. Xu, Yongqin Gao, S. Christley, and G. Madey. "A Topological Analysis of the Open Souce Software Development Community." In: *Proceedings of the 38th Annual Hawaii International Conference on System Sciences* (2005), 198a–198a (cit. on p. 65).

ONLINE RESSOURCES

[27] S.P. Borgatti, M. G. Everett, and L. C. Freeman. *UCINET 6 for Windows: Software for Social Network Analysis. Harvard, MA, Analytic Technologies*. 2002 (cit. on p. 9).

[28] Jaime Niswonger. *Why node.js? - Introduction and Analysis*. Website. Available online at https://keyholesoftware.com/company/creations/white-papers/why-node-js/; visited on October 30th 2020. 2016 (cit. on p. 35).

[29] DTP Team. *Data Transfer Project Overview and Fundamentals*. Website. Available online at https://datatransferproject.dev/dtp-overview.pdf; visited on October 28th, 2020. July 2018 (cit. on p. 33).

[30] GitHub Docs Team. *About organizations*. Website. Available online at https://docs.github.com/en/free-pro-team\spacefactor\@m{}latest/github/setting-up-and-managing-organizations-and-teams/about-organizations; visited on November 15th, 2020. 2020 (cit. on p. 11).

[31] GitHub Docs Team. *Adding notes to a project board*. Website. Available online at https://docs.github.com/en/github/managing-your-work-on-github/adding-notes-to-a-project-board#converting-a-note-to-an-issue; visited on September 10th, 2020. 2020 (cit. on p. 71).

[32] GitHub Docs Team. *Blocking a user from your organization*. Website. Available online at https://docs.github.com/en/github/building-a-strong-community/blocking-a-user-from-your-organization#blocking-a-user-in-a-comment; visited on September 12th, 2020. 2020 (cit. on p. 77).

[33] GitHub Docs Team. *Changing the state of a pull request*. Website. Available online at https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/changing-the-stage-of-a-pull-request#converting-a-pull-request-to-a-draft; visited on September 10th, 2020. 2020 (cit. on p. 71).

[34] GitHub Docs Team. *Creating a pull request*. Website. Available online at https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/creating-a-pull-request; visited on September 9th, 2020. 2020 (cit. on pp. 70, 72).

[35] GitHub Docs Team. *Dismissing a pull request review*. Website. Available online at https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/dismissing-a-pull-request-review; visited on September 12th, 2020. 2020 (cit. on p. 75).

[36] GitHub Docs Team. *Issue event types: Added To Project*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#added_to_project; visited on September 10th, 2020. 2020 (cit. on p. 69).

[37] GitHub Docs Team. *Issue event types: Assigned*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#assigned; visited on September 10th, 2020. 2020 (cit. on p. 69).

[38] GitHub Docs Team. *Issue event types: Automatic Base Change Failed*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#automatic_base_change_failed; visited on September 9th, 2020. 2020 (cit. on p. 69).

[39] GitHub Docs Team. *Issue event types: Automatic Base Change Succeeded*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#automatic_base_change_succeeded; visited on September 9th, 2020. 2020 (cit. on p. 70).

[40] GitHub Docs Team. *Issue event types: Base Ref Changed*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#base_ref_changed; visited on September 10th, 2020. 2020 (cit. on p. 70).

[41] GitHub Docs Team. *Issue event types: Closed*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#closed; visited on September 10th, 2020. 2020 (cit. on p. 70).

[42] GitHub Docs Team. *Issue event types: Commented*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#commented; visited on September 10th, 2020. 2020 (cit. on p. 70).

[43] GitHub Docs Team. *Issue event types: Committed*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#committed; visited on September 10th, 2020. 2020 (cit. on p. 70).

[44] GitHub Docs Team. *Issue event types: Connected*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#connected; visited on September 10th, 2020. 2020 (cit. on p. 70).

[45] GitHub Docs Team. *Issue event types: Convert To Draft*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#convert_to_draft; visited on September 10th, 2020. 2020 (cit. on p. 71).

[46] GitHub Docs Team. *Issue event types: Converted Note To Issue*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#converted_note_to_issue; visited on September 10th, 2020. 2020 (cit. on p. 71).

[47] GitHub Docs Team. *Issue event types: Cross Referenced*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#cross-referenced; visited on September 10th, 2020. 2020 (cit. on p. 71).

[48] GitHub Docs Team. *Issue event types: Demilestoned*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#demilestoned; visited on September 10th, 2020. 2020 (cit. on p. 71).

[49] GitHub Docs Team. *Issue event types: Deployed*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#deployed; visited on September 14th, 2020. 2020 (cit. on p. 72).

[50] GitHub Docs Team. *Issue event types: Deployment Environment Changed*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#deployement_environment_changed; visited on September 14th, 2020. 2020 (cit. on p. 72).

[51] GitHub Docs Team. *Issue event types: Disconnected*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#disconnected; visited on September 10th, 2020. 2020 (cit. on p. 72).

[52] GitHub Docs Team. *Issue event types: Head Ref Deleted*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#head_ref_deleted; visited on September 14th, 2020. 2020 (cit. on p. 72).

[53] GitHub Docs Team. *Issue event types: Head Ref Restored*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#head_ref_restored; visited on September 14th, 2020. 2020 (cit. on p. 72).

[54] GitHub Docs Team. *Issue event types: Labeled*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#labeled; visited on September 14th, 2020. 2020 (cit. on p. 72).

[55] GitHub Docs Team. *Issue event types: Locked*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#locked; visited on September 14th, 2020. 2020 (cit. on p. 73).

[56] GitHub Docs Team. *Issue event types: Marked As Duplicate*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#marked-as-duplicate; visited on September 10th, 2020. 2020 (cit. on p. 73).

[57] GitHub Docs Team. *Issue event types: Mentioned*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#mentioned; visited on September 14th, 2020. 2020 (cit. on p. 73).

[58] GitHub Docs Team. *Issue event types: Merged*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#merged; visited on September 10th, 2020. 2020 (cit. on p. 73).

[59] GitHub Docs Team. *Issue event types: Milestoned*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#milestoned; visited on September 14th, 2020. 2020 (cit. on p. 73).

[60] GitHub Docs Team. *Issue event types: Moved Columns In Project*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#moved_columns_in_project; visited on September 14th, 2020. 2020 (cit. on p. 73).

[61] GitHub Docs Team. *Issue event types: Pinned*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#pinned; visited on September 10th, 2020. 2020 (cit. on p. 74).

[62] GitHub Docs Team. *Issue event types: Ready For Review*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#ready_for_review; visited on September 14th, 2020. 2020 (cit. on p. 74).

[63] GitHub Docs Team. *Issue event types: Referenced*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#referenced; visited on September 14th, 2020. 2020 (cit. on p. 74).

[64] GitHub Docs Team. *Issue event types: Removed From Project*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#removed_from_project; visited on September 14th, 2020. 2020 (cit. on p. 74).

[65] GitHub Docs Team. *Issue event types: Renamed*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#renamed; visited on September 14th, 2020. 2020 (cit. on p. 74).

[66] GitHub Docs Team. *Issue event types: Reopened*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#reopened; visited on September 14th, 2020. 2020 (cit. on p. 75).

[67] GitHub Docs Team. *Issue event types: Review Dismissed*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#review_dismissed; visited on September 14th, 2020. 2020 (cit. on p. 75).

[68] GitHub Docs Team. *Issue event types: Review Request Removed*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#review_request_removed; visited on September 14th, 2020. 2020 (cit. on p. 75).

[69] GitHub Docs Team. *Issue event types: Review Requested*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#review_requested; visited on September 14th, 2020. 2020 (cit. on p. 75).

[70] GitHub Docs Team. *Issue event types: Reviewed*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#reviewed; visited on September 14th, 2020. 2020 (cit. on p. 75).

[71] GitHub Docs Team. *Issue event types: Subscribed*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#subscribed; visited on September 14th, 2020. 2020 (cit. on p. 75).

[72] GitHub Docs Team. *Issue event types: Transferred*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#transferred; visited on September 14th, 2020. 2020 (cit. on p. 75).

[73] GitHub Docs Team. *Issue event types: Unassigned*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#unassigned; visited on September 14th, 2020. 2020 (cit. on p. 76).

[74] GitHub Docs Team. *Issue event types: Unlabeled*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#unlabeled; visited on September 14th, 2020. 2020 (cit. on p. 76).

[75] GitHub Docs Team. *Issue event types: Unlocked*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#unlocked; visited on September 14th, 2020. 2020 (cit. on p. 76).

[76] GitHub Docs Team. *Issue event types: Unmarked as duplicate*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#unmarked_as_duplicate; visited on September 12th, 2020. 2020 (cit. on p. 76).

[77] GitHub Docs Team. *Issue event types: Unpinned*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#unpinned; visited on September 12th, 2020. 2020 (cit. on p. 76).

[78] GitHub Docs Team. *Issue event types: Unsubscribed*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#unsubscribed; visited on September 14th, 2020. 2020 (cit. on p. 76).

[79] GitHub Docs Team. *Issue event types: User Blocked*. Website. Available online at https://docs.github.com/en/developers/webhooks-and-events/issue-event-types#user_blocked; visited on September 14th, 2020. 2020 (cit. on p. 77).

[80] GitHub Docs Team. *Limiting interactions in your repository*. Website. Available online at https://docs.github.com/en/github/building-a-strong-community/limiting-interactions-in-your-repository; visited on September 25th, 2020. 2020 (cit. on p. 12).

[81] GitHub Docs Team. *Locking conversations*. Website. Available online at https://docs.github.com/en/github/building-a-strong-community/locking-conversations; visited on September 10th, 2020. 2020 (cit. on pp. 73, 76).

[82] GitHub Docs Team. *Merging a pull request*. Website. Available online at https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/merging-a-pull-request; visited on September 10th, 2020. 2020 (cit. on pp. 13, 73).

[83] GitHub Docs Team. *Pinning an issue to your repository*. Website. Available online at https://docs.github.com/en/free-pro-team\spacefactor\@m{}latest/github/managing-your-work-on-github/pinning-an-issue-to-your-repository; visited on November 12th, 2020. 2020 (cit. on p. 74).

[84] GitHub Docs Team. *Project board permissions for an organization: Permissions overview*. Website. Available online at https://docs.github.com/en/github/setting-up-and-managing-organizations-and-teams/project-board-permissions-for-an-organization#permissions-overview; visited on September 10th, 2020. 2020 (cit. on p. 71).

[85] GitHub Docs Team. *Project board permissions for an organization*. Website. Available online at https://docs.github.com/en/github/setting-up-and-managing-organizations-and-teams/project-board-permissions-for-an-organization; visited on September 9th, 2020. 2020 (cit. on pp. 69, 73, 74).

[86] GitHub Docs Team. *Repository permission levels for an organization*. Website. Available online at https://docs.github.com/en/github/setting-up-and-managing-organizations-and-teams/repository-permission-levels-for-an-organization#repository-access-for-each-permission-level; visited on September 9th, 2020. 2020 (cit. on pp. 12, 13, 69–76).

[87] GitHub Docs Team. *Requesting a pull request review*. Website. Available online at https://docs.github.com/en/github/collaborating-with-issues-and-pull-requests/requesting-a-pull-request-review; visited on September 12th, 2020. 2020 (cit. on p. 75).

[88] GitHub Docs Team. *Setting base permissions for an organization*. Website. Available online at https://docs.github.com/en/github/setting-up-and-managing-organizations-and-teams/setting-base-permissions-for-an-organization; visited on September 25th, 2020. 2020 (cit. on p. 13).

[89] GitHub Docs Team. *Transferring an issue to another repository*. Website. Available online at https://docs.github.com/en/github/managing-your-work-on-github/transferring-an-issue-to-another-repository; visited on September 12th, 2020. 2020 (cit. on p. 76).

[90] Nextcloud Team. *Nextcloud Solution Architecture*. Website. Available online at https://nextcloud.com/whitepapers/; visited on October 29th, 2020. 2018 (cit. on p. 35).