Automatic Core-Developer Identification on GitHub: A Validation Study

THOMAS BOCK, Saarland University, Saarland Informatics Campus, Germany
NILS ALZNAUER, Saarland University, Saarland Informatics Campus, Germany
MITCHELL JOBLIN, Siemens AG, and Saarland University, Saarland Informatics Campus, Germany
SVEN APEL, Saarland University, Saarland Informatics Campus, Germany

Many open-source software (OSS) projects are self-organized and do not maintain official lists with information on developer roles. So, knowing which developers take core and maintainer roles is, despite being relevant, often tacit knowledge. We propose a method to *automatically* identify core developers based on role permissions of privileged events triggered in GitHub issues and pull requests. In an empirical study on 25 GitHub projects, (1) we validate the set of automatically identified core developers with a sample of project-reported developer lists, and (2) we use our set of identified core developers to assess the accuracy of state-of-the-art unsupervised developer classification methods. Our results indicate that the set of core developers, which we extracted from privileged issue events, is sound and the accuracy of state-of-the-art unsupervised classification methods depends mainly on the data source (commit data vs. issue data) rather than the network-construction method (directed vs. undirected, etc.). In perspective, our results shall guide research and practice to choose appropriate *unsupervised* classification methods, and our method can help create reliable ground-truth data for training *supervised* classification methods.

CCS Concepts: • **Software and its engineering** → **Software creation and management**; **Collaboration in software development**; **Open source model**.

Additional Key Words and Phrases: open-source software projects, developer classification, developer networks

1 INTRODUCTION

Coordination among software developers is critical to ensure software quality and to drive software evolution [20, 44, 53, 55]. This holds especially for open-source software (OSS) projects, in which volunteers can participate and developers are globally distributed, often not knowing each other personally [5, 43]. With increasing project size and popularity, developers who hold leadership roles (i.e., who take care of the project's health and are highly involved in its long-term maintenance) are crucial for a thriving evolution of the project [23, 61, 99, 102]. As opposed to closed-source software projects, where leadership positions are typically established by mandated organization structures and well-determined within the organization, OSS projects often do not provide explicit, publicly recognizable information regarding group structures and project responsibilities. Still, in OSS projects, hierarchical group structures among developers do exist [87, 103, 107], but they are typically not determined by a centralized authority but rather emerge following principles of self-organization [28, 104]. So, despite the lack of predefined hierarchical group structures in OSS projects, certain developers take particular responsibilities in the project, such as performing maintenance tasks or implementing core functionality. Such developers, often

Authors' addresses: Thomas Bock, Saarland University, Saarland Informatics Campus, Saarbrücken, Germany; Nils Alznauer, Saarland University, Saarland Informatics Campus, Saarbrücken, Germany; Mitchell Joblin, Siemens AG, and Saarland University, Saarland Informatics Campus, Munich / Saarbrücken, Germany; Sven Apel, Saarland University, Saarland Informatics Campus, Saarbrücken, Germany.

1049-331X/2023/4-ART $15.00
https://doi.org/10.1145/3593803

called *maintainers* or *core developers* [46, 48, 66, 68], become crucial to these projects, as they hold key knowledge about source code and software architecture [78] and shape the character and culture of the project [1, 92, 108], especially when OSS projects grow fast in terms of developers and amount of source code [36]. As a prominent example, the Linux kernel maintains a public list of maintainers, who are responsible for specific subsystems and who have deep knowledge about the project [107]. Using this list, external and internal developers can find out whom to contact for any matters. Unfortunately, many OSS projects do not maintain such a public, curated list. Especially for newcomers in a software project, but also for developers that are not consistently active (also called *peripheral developers* [27, 48, 66, 68]), it is important to know who is playing the role of a core developer, who is making decisions and maintaining the code base, and who will finally accept or reject code contributions [62, 84]. In well-organized projects, this might not be an actual problem for newcomers, as one can easily check who has recently reviewed or merged pull requests. Some projects even use bots to automatically assign developers for reviewing a new pull request [94]. However, in projects that are badly organized and which do not use such well-structured code-review processes, it might be not that easy to find out whom to contact in the case of questions, for instance. Knowing how developers organize is also important for companies, to decide whether and how to invest in a project or how to efficiently contribute to an OSS project [16, 33, 106, 109]. For researchers, knowing about core developers also aids the investigation of a project's hierarchical structure, which helps avoid high developer turnover and identify organizational smells that could endanger project success [19, 24, 29, 42, 55, 75, 86].

There is a growing corpus of research that attempts to *automatically* extract information about the core developers of a project. A state-of-the-art approach is to construct a developer network from various data sources (e.g., commits, e-mails, issue comments) and to apply network-based centrality metrics. The assumption is that core developers are responsible for about 80% of the contributions and activities in the project [48, 49, 51, 62, 65, 101, 105], attaining central positions within the network due to the high number of contributions and many interactions with other developers. While this is an appealing approach, the actual performance of such classification methods on the identification of core developers is largely unclear, especially as there are various ways on how to construct developer networks and also many, possibly contradicting classification methods can be applied thereafter.

Therefore, we devise a method to *automatically* derive a set of core developers and maintainers from privileged events in GitHub issue discussions and pull requests. The rationale is that GitHub permits triggering certain events (e.g., merging a pull request) only to user accounts that have been assigned to a certain role. Nevertheless, these privileged events cannot be used to identify potential candidates for future core developers before they get any privileges in the project and also not for projects that user other social coding platforms than GitHub issues. Therefore, other classification methods for identifying core developers are still essential, which is why it is also our goal to assess the accuracy of the state-of-the-art classification methods. In an empirical study on 25 widely-used and well-known GitHub projects, we (1) validate our automatically derived set of core developers based on privileged issue events with publicly available, project-reported lists of maintainers or core developers for projects which provide such lists, and we (2) use our set of core developers to assess the accuracy of various state-of-the-art unsupervised developer classification methods based on well-established count-based metrics (e.g., commit count) or network-based metrics (e.g., degree centrality).

Our results indicate that our automatically identified set of core developers is sound and that the accuracy of state-of-the-art unsupervised developer classification methods depends mainly on the data source (commit data vs. issue data) rather than the network-construction method (e.g., directed vs. undirected). Combining issue data and commit data behaves similarly to just using issue data. Our results hold the potential to help researchers and OSS communities choose the appropriate *unsupervised* classification method for identifying core developers or potential maintainers.

Moreover, our method can help create reliable ground-truth data for training *supervised* classification methods, as—to the best of our knowledge—the lack of sufficient volumes of ground-truth data has prevented supervised

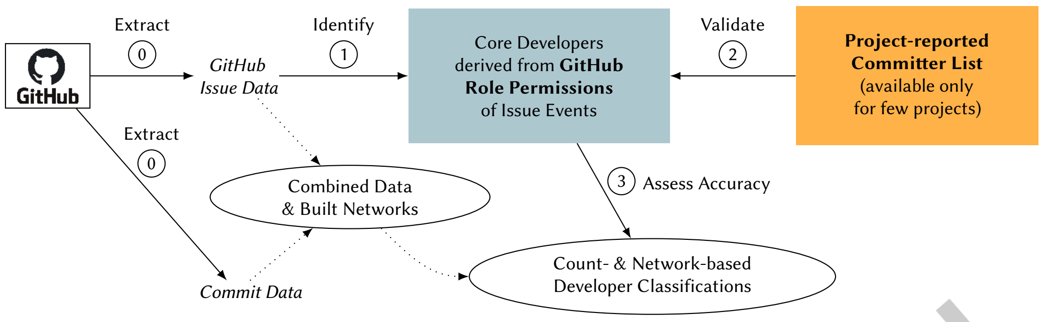
GitHub
Extract
⓪
GitHub Issue Data
Identify
①
Core Developers derived from **GitHub Role Permissions** of Issue Events
Validate
②
Project-reported Committer List (available only for few projects)
Extract
⓪
Combined Data & Built Networks
③ Assess Accuracy
Count- & Network-based Developer Classifications
Commit Data

Fig. 1. Overview of our approach: (⓪) extract data from GitHub, (①) identify a set of core developers based on the events in issues and pull requests, (②) validate it with project-reported lists, (③) assess the accuracy of state-of-the-art unsupervised classification methods.

learning methods from being developed. Although we devise a method to automatically derive a set of core developers from privileged events in GitHub issues, there is still a need for supervised classification methods: On the one hand, it is more tedious to get the necessary data for the extraction of privileged events than for the state-of-the-art count-based or network-based classification methods, as privileged events may not be available for projects that use other social coding platforms than GitHub. On the other hand, supervised classification can be used to identify potential future core developers and maintainers from activity data before the developers get any privileges in the project. Of course, becoming a core developer might not only be a matter of participation, but also a matter of trust [3, 17, 83]. As trust, however, cannot be quantified in the accessible data, using activity data can be a first step towards finding potential candidates, especially since accepted commits and intensive involvements in issue discussions can be used as a first indicator for the trustworthiness of a candidate [31].

In summary, we make the following contributions:

- An overview of literature on how core developers have been identified in previous work (see Tables 1 and 2).
- A method to *automatically* identify a set of core developers based on role permissions of events triggered in GitHub issues and pull requests (see step ① in Figure 1).
- A validation of our automatically identified set of core developers based on privileged issue events with official, publicly available maintainer or committer lists (see step ② in Figure 1).
- An assessment of the accuracy of several state-of-the-art *unsupervised* developer-role classification methods that use count- and network-based metrics (see step ③ in Figure 1). We found that network-construction methods do not make a substantial difference, whereas the data source plays a more vital role: Classification methods using commit data perform better than classification methods using issue data. Combining issue and commit data behaves similarly to just using issue data, as the combination is dominated by the issue data.
- A small explorative experiment on how our automatically identified set of core developers can be used for training *supervised* classification methods, from which we derive two hypotheses that should serve as a starting point for future research on this topic.
- A replication package including pseudonymized raw data, classification data, tooling, and scripts on a supplementary website:
https://se-sic.github.io/paper-developer-classifications/ and https://zenodo.org/record/7775882 .

2 BACKGROUND & RELATED WORK

In this section, we provide background information and related work on determining developers' roles and on how developers can be classified into core and peripheral roles.

2.1 Developer Roles

As developers in OSS projects undertake different kinds of tasks and have different roles, Nakakoji et al. [68] proposed the so-called "onion model" distinguishing eight different roles of community members: There are several roles for project users (passive user, reader, bug reporter), who do not contribute to the project's source code. Five roles are directly related to source-code contributions: bug fixer, peripheral developer, active developer, core member (also called maintainer), and project leader (who are mostly the project initiators). Xu et al. [100] proposed a similar model composed of four developer roles. As the distinction into four or five different developer roles is rather fine-grained and their boundaries are blurred, researchers often conflate them into just two roles: Whereas bug fixers and peripheral developers only contribute occasionally and sporadic, according to the "onion model", active developers, core members, and project leaders contribute regularly. As a consequence, the occasionally and sporadically contributing developers are called *peripheral* developers, whereas the remaining ones are called *core* developers or *maintainers* [27, 30, 48, 49, 66, 89]. When automatically classifying developers, often only these two roles are considered (see Section 2.3), so we use only these two roles in our study. Researchers sometimes additionally consider the role of one-time contributors (also called newcomers), who contribute only once to a project (e.g., provide a single bug fix) [25, 57]. One-time contributors, though, are considered to be part of the group of peripheral developers [58, 73, 81].

In addition to the above mentioned developer roles, there are also other approaches on how developer roles can be defined. For example, Cheng et al. [23] distinguish between developmental core developers and collaborative core developers. However, such a distinction is not disjoint, as highly active developers may also collaborate with many others. Constantino et al. [25] differentiate between project roles and committer roles. Whereas the project roles represent various organizational and potentially overlapping tasks (developer, maintainer, team leader, project promoter, reviewer, or coordinator), the committer roles encompass what we consider as core and peripheral developer roles. Montandon et al. [67] distinguish technical roles of GitHub users across projects, driven by the variety of technical tasks a developer mostly takes across all the projects a developer contributes to (e.g., contribute to the frontend, to the backend, or to the continuous integration), whereas we study the activity role of a developer within a specific project. Instead of defining concrete developer roles, Bock et al. [9] look at groups of developers that form up around certain tasks, and they analyze the importance and stability of these groups. In their approach, they identify groups of developers via canonical tensor decomposition on tensors that model the communication and collaboration behavior of developers over time. This way, they determine the importance of a developer based on the importance of the latent tasks in which the developer is involved.

Wang et al. [92] define the role of an elite developer as a developer having write permission in a GitHub project. They identify elite developers within time ranges of 3 months based on performed tasks that require write permission. This approach is similar to how we identify core developers. However, there are a couple of differences between their approach and our approach: Whereas they access GitHub's event API to gather the performed tasks of a project (e.g., whether somebody has forked a project or starts watching a project) and merge it with additional information from other data sources, we directly access GitHub's issues API, which contains more detailed information on the events that happen specifically in issues and pull requests (e.g., a comment was added, an issue was labeled, a pull-request was merged, etc.). Another difference is that they needed to classify the gathered data into a complex taxonomy of event types (communicative, organizational, typical, supportive), whereas we just use GitHub's official description of all the possible issue event types, which contains information on the user permission that is needed to trigger a specific issue event. Finally, while they use fixed time ranges of 3 months, we study the time difference between certain issue events and investigate different times ranges between 3 and 12 months.

2.2 User Permissions on GitHub

As we identify core developers based on user permissions, we briefly describe which user roles and permission levels exist on GitHub. First of all, GitHub distinguishes between *organization* (shared account across many projects) and *user* accounts. Repositories (also called projects) can be created from either an organization or a user account. Each organization on GitHub has, at least, one organization owner (which is a normal user). Organization owners and project admins can set the user permission individually for each project and user. At the project level, GitHub users can have one of the following permissions:[1]

- ***Read***: default permission for any user seeing a project, "recommended for non-code contributors who want to view or discuss your project"[1]. Users with read permission can open issues or pull-requests, submit reviews on pull requests, write comments, and close issues or pull requests that have been opened by themselves.
- ***Triage***: "recommended to proactively manage issues and pull requests without write access"[1]. In addition to read permission, a user with triage permission can set labels, close and reopen issues, assign issues to users, and request reviews from users.
- ***Write***: "recommended for contributors who actively push to your project"[1]. In addition to triage permission, a user with write permission can push to a repository (i.e., directly push source code to the repository), merge pull requests, publish releases, and submit reviews that "affect a pull request's mergeability"[1].
- ***Maintain***: "recommended for project managers who need to manage the repository without access to sensitive and destructive actions"[1]. In addition to write permission, a user with maintain permission can protect branches, decide on how pull requests can be merged, and restrict which other users can open issues or pull requests, etc.
- ***Admin***: "recommended for people who need full access to the project, including sensitive and destructive actions like managing security or deleting repository"[1]. When creating a project, the user creating the project has admin permission. Organization owners implicitly have admin permissions on all projects of the organization.

In our study, we make use of these permissions to automatically identify a set of core developers by mapping GitHub's user permissions to our developer roles (see Section 3.2). Unfortunately, it is not publicly accessible which user has which permission in a GitHub project. Instead, we need to infer the users' permissions from certain actions GitHub users perform on the issues and pull requests of a specific project.

2.3 Classification of Developer Roles

As opposed to our permission-based approach to automatically classify developers into core and peripheral, researchers have developed and investigated various unsupervised, manually tuned methods. To obtain an overview of how core developers have been identified in previous work, we performed a series of searches on multiple different search engines (GoogleScholar, IEEEexplore, ACM Digital Library) using the search terms "core developer" and "identification of core developers", without restricting venue and publication year. We manually checked the papers of the search results and looked for core-developer identification techniques that were described, mentioned, or referenced in these papers. Moreover, we collected such techniques also from papers that have been referenced in these papers in a backward-search fashion. In what follows, we summarize the core-developer identification techniques that we have collected from the literature. Additionally, we manually extracted the used data sources and classification metrics from all the collected papers. In Table 1, we provide an overview of the data sources and network types that have been used in the literature to identify core developers; in Table 2 we summarize the used metrics.

[1]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 (accessed: 2020-09-09)

Mockus et al. [66] proposed that highly active developers (in terms of the number of contributions) should be considered as core developers. They investigated the projects Mozilla and Apache Web Server and identified that only a small number of 10–15 developers are responsible for around 80% of the source-code contributions. Dinh-Trong and Bieman [34] replicated their study on FreeBSD and showed that even more than 15 developers contribute 80% of the code base. Follow-up research showed that Zipf's law holds for the number of commits that are authored by a developer and, consequently, the top 20% of the developers are responsible for 80% of the commits [30]. Using this 80% threshold is a commonly used approach when classifying developers based on their commit count [2, 24, 30, 34, 37, 52, 66, 78–80, 89, 101]. For this reason, we also use this threshold in our validation study for the developer classification based on commit count when we compare our approach with the state-of-the-art classification methods (see Section 3.4). Valiev et al. [91] even used a threshold of 90% of contributions per month, which was selected based on empirical observations on the Python Package Index ecosystem. Coelho et al. [24] and Ferreira et al. [37] extended this approach by adding the additional restriction that a core developer has to have authored, at least, 5% of the total number of commits in the project. Instead of considering the number of commits, Canedo et al. [18] required a core developer to have authored, at least, 50% of the files in the project. Nonetheless, relying on count-based operationalizations for developer classification provides only limited insights into organizational matters of OSS projects, as relationships among developers (possibly varying over time) are neglected [48].

To incorporate relationships among developers, Crowston et al. [30] suggested three different approaches on how to classify developers into core and peripheral: For some of their analyzed projects, they found project-reported lists on the projects' websites, containing formal roles of developers (e.g., who is allowed to check-in source code to the version-control system). This approach is similar to how we validate our set of core developers (see Section 3.3). Using project-reported lists to determine core developers has also been adopted by other researches [29, 54, 93]. However, such project-reported lists are only available for few projects. In their second approach, Crowston et al. [30] considered developers to be core if they contribute, at least, one third of the total number of comments on bug-tracker data from SourceForge. This approach is similar to the count-based classification mentioned above and is based on the sheer number of posted comments. In their third approach [30], they built communication networks on bug reports and applied graph partitioning algorithms, treating the tightly interconnected group as core developers. All three approaches led to different results as the project-reported list seemed to be incomplete and count-based and network-based classification focus on different characteristics of core developers.

Network-based developer classification has been gaining considerable momentum in software-engineering research. De Souza et al. [32] created networks based on developers' contributions to the same modules. Bird et al. [7] built networks based on mailing-list communication, and they used basic social network analysis techniques to identify developers' roles [5, 6, 8]. Licorish and MacDonell [59, 60] used networks built from the communication on tasks in IBM's Rational Jazz development environment and considered a developer to be core if they had contributed to the communication of, at least, one third of the tasks. Oliva et al. [69, 70] identified core developers in developer networks based on mailing lists and version-control systems. They used centrality metrics combined with a quartile analysis to determine core developers, but they investigated only a short time-period of a small project. We, thus, investigate a history of several years for 25 projects. Moreover, in our validation study, we build developer networks based on version-control systems and communication in issues, not based on mailing lists.

Joblin et al. [48] constructed developer networks from the version-control system data and mailing lists of 10 OSS projects. They applied the network centrality metrics degree centrality, eigenvector centrality, and hierarchy centrality, which capture structural differences in the relationships among developers. To evaluate the classification outcomes, they conducted a survey among 166 developers of the examined projects. According to the survey, the network-based developer classifications outperformed the simpler count-based developer classifications. Notably, there is a substantial corpus of work that relies on network-based developer classifications

Table 1. Classification data used in the literature

	Data / Network Type	Papers
Count-based	Commits	[2, 18, 24, 37, 77–80, 89, 91, 101]
	E-mails	[54]
	Issues	[30, 88]
	Commits, e-mails	[48, 52, 69, 70]
	Commits, e-mails, issues	[34, 66]
Network-based	Cochange	[23, 49, 63, 71, 72]
	E-mail	[6, 50, 72, 82]
	Issue	[4, 27, 30, 50, 59, 60, 72]
	Cochange, e-mail	[5–7, 48, 50, 69, 70, 105]
	Cochange, issue	[13, 35, 50, 88]

Table 2. Classification metrics used in the literature

	Metrics	Papers
Count-based	Commits	[2, 24, 32, 37, 48, 69, 77, 79, 80, 89, 91, 101]
	Lines of code (LOC)	[48, 52, 66, 78, 101]
	Authored files	[18]
	E-mails	[54]
	Modification requests	[66]
	Issue comments	[30]
	Pull-request comments	[88]
Network-based	Degree centrality	[4–6, 13, 23, 27, 32, 35, 48, 63, 70, 72, 82, 90, 105]
	Betweenness centrality	[4–6, 13, 23, 27, 35, 63, 70, 82, 90, 105]
	Eigenvector centrality	[4, 13, 35, 48, 50, 69, 70, 82]
	Closeness centrality	[4, 13, 23, 35, 63, 70, 82]
	PageRank	[4, 13, 35, 105]
	Clustering coefficient	[48, 49, 63]
	Hierarchy centrality	[48–50]
	Density	[59, 60]
	Modularity	[7, 49]
	Eccentricity	[4]
	Graph partitioning	[30]
	HITS	[105]
	Scale freeness	[49]

to investigate characteristics of core and peripheral developers and organizational structures in OSS projects [4, 13, 23, 35, 49, 50, 63, 71, 72, 82, 88, 90]. Naturally, this work depends on the accuracy of the classifications, which is exactly the motivation of our validation study, in which we assess the accuracy of a selection of the above mentioned classification methods.

Zhang et al. [105] investigated whether the social-network metrics node degree (i.e., degree centrality), PageRank [56] (a variant of eigenvector centrality), HITS [39] (an algorithm to detect hubs and authorities), and betweenness of nodes in the network are capable of identifying core developers in OSS projects. They assessed the accuracy of their classification methods against a self-created ground truth based on “the right to post messages” on the developer mailing list of the project ArgoUML, considering only developers that had regular contributions to source code and mailing list. All four network metrics performed similarly well in detecting core developers, having a recall of more than 60% and a precision of about 60%. Similarly to their study, we also assess the accuracy of classification metrics using a ground truth that is based on the permissions to perform privileged events on GitHub issues. We improve over their study by investigating a variety of GitHub projects, constructing different types of developer networks, and using network-based centrality metrics that have been used in state-of-the-art research on developer classification.

3 METHODOLOGY

As illustrated in Figure 1 (page 3), our validation study proceeds in four steps: In the initial step (⓪), we extract commit and issue data from GitHub. Then (①), we identify the developers who have triggered a privileged issue event. Following this (②), we validate this set of developers with official, project-reported committer lists. Finally (③), we perform unsupervised developer classifications and assess their accuracy.

3.1 Data Extraction

We use the tool Codeface[2] [48, 49, 51] to extract commit metadata from Git, including author name, e-mail address, author date, and the names of the changed files of a commit.

For extracting issue and pull-request data from GitHub, we developed the tool GitHubWrapper[3], which queries issue metadata (incl. review data, review comments, and all other comments of a pull request) from GitHub's official REST API[4]. Review data, review comments, and the remaining comments for a pull request are separately crawled (as they need to be accessed through different interfaces of GitHub's API) and then combined with the remaining issue event data. Our tool also performs additional postprocessing steps to unify the data (e.g., correctly determine the actual actor of a "subscribed" event, as on GitHub such an event can either be actively triggered by a user subscribing themselves to an issue or, passively, by being mentioned by someone else) or gather additional information (e.g., extracting names and e-mail addresses from referenced commits). The extracted issue metadata contain information on which user has triggered which event (commented, labeled, merged, etc.), on which issue or pull request, and at which point in time. Notice that GitHub internally treats pull requests (PRs) as a special form of issues. For that reason, when we talk about issues this always also includes pull requests.

Using the tool codeface-extraction[5], we merge the commit and issue data to match developers who use the same name or the same e-mail address.[6] Developers found in only one of the data sources are kept, though. To match developers via names and e-mail addresses, we use the disambiguation heuristic of Oliva et al. [69], which has been proved to be accurate [96], and perform additional sanity checks and manual corrections.

More and more GitHub projects use automatic bots which, for instance, submit comments to issues or review pull requests. Bots often also close abandoned issues, execute automatic code refactorings, run continuous-integration tasks, or perform similar tasks [15, 41, 94, 95, 98]. Previous research has shown that about 20% of issue comments are posted by bots and about 31% of pull requests involve bots [40]. Not to distort developer-network characteristics and core-developer identification, we must detect and remove bots' activities from the extracted issue data. Whereas some bots are labeled by GitHub to be bots or use a "bot" suffix in their usernames, research has found that many bots are not labeled as such and also human users can use the "bot" suffix, which makes

[2]https://github.com/se-sic/codeface/ (accessed: 2022-03-14)
[3]https://github.com/se-sic/GitHubWrapper/
[4]https://docs.github.com/en/rest/ (accessed: 2022-03-14)
[5]https://github.com/se-sic/codeface-extraction/ (accessed: 2022-03-14)
[6]As some GitHub users keep their real names or e-mail addresses private, we only use their public usernames if no name or e-mail is available on GitHub.

Table 3. GitHub issue events and the role permissions needed to be able to trigger them

Privileged Events (*write*, *maintain*, or *admin* permission)	Extended Events (at least, *triage* permission)	Common Events (at least, *read* permission)
`added_to_project`, `converted_note_to_issue`, `deployed`, `deployment_environment_changed`, `locked`, `merged`, `moved_columns_in_project`, `pinned`, `removed_from_project`, `review_dismissed`, `transferred`, `unlocked`, `unpinned`, `user_blocked`	`assigned`, `demilestoned`, `labeled`, `marked_as_duplicate`, `milestoned`, `unassigned`, `unlabeled`, `unmarked_as_duplicate`	`automatic_base_change_failed`, `automatic_base_change_succeeded`, `base_ref_changed`, `closed`, `comment_deleted`, `commented`, `committed`, `connected`, `convert_to_draft`, `created`, `cross_referenced`, `disconnected`, `head_ref_deleted`, `head_ref_restored`, `mentioned`, `ready_for_review`, `referenced`, `referenced_by`, `renamed`, `reopened`, `review_request_removed`, `review_requested`, `reviewed`, `subscribed`, `unsubscribed`

bot detection a non-trivial task [40]. For that reason, Golzadeh et al. [40] have developed the tool BoDeGHa to automatically detect bots based on the total number of comments and commenting patterns. Unfortunately, many projects use predefined issue or pull-request comment templates[7], making BoDeGHa misclassify human users as bots when they mostly use templates or post stereotyped comments. As a consequence, after using BoDeGHa for automatic bot detection on users that have posted, at least, 2 000 comments in a project, we perform some manual adjustments. For example, we classify 83 users that are widely-used bots as a bot even if the automatic bot detection did not classify the user to be a bot (e.g., the "lockbot", which is a bot that automatically locks issues but usually does not post comments and, therefore, is not detectable by BoDeGHa). In particular, we manually checked all usernames that had a "bot" substring (such as the "lockbot") whether they are marked as a bot by GitHub. If so, we added them to the set of bots identified by BoDeGHa. After bot detection, we remove all bot-triggered events from the extracted GitHub issue data.

3.2 Identifying Core Developers based on Issue Events

To identify core developers, we analyze the events that appear in the above extracted issue data. For each of the events, we look up in GitHub's official event documentation[8] which user permission on GitHub is needed to trigger the event (e.g., write permission is needed to merge a pull request). This way, we form three categories of events: *Common events*, which can be triggered by everyone (e.g., write a comment), *extended events* that can be triggered only when having, at least, triage permission (e.g., apply a label), and *privileged events* that can be triggered only when having, at least, write permission (e.g., merge a pull request). In Table 3, we provide an overview of all events and the categories we assigned them to.

As a next step, we analyze which user has triggered which event. Each user who has triggered a privileged event is considered a core developer (at least, in the time range that contains the event). All remaining users are considered peripheral developers. We base this decision on the fact that events that require, at least, write permission are either related to critical tasks for project maintenance (blocking users or locking issues) or to decision making tasks that require deep project knowledge (e.g., merging or rejecting a pull request), and therefore distinguish a core developer.[9]

[7]There are various different means of how issue templates can be used and enforced by a project, which also varies during project evolution. One example can be found in the official documentation of GitHub: https://docs.github.com/en/github/building-a-strong-community/about-issue-and-pull-request-templates/ (accessed: 2022-03-14)

[8]GitHub provides a list of possible events here: https://docs.github.com/en/developers/webhooks-and-events/issue-event-types/ (accessed: 2020-09-09)

[9]We have conducted a sensitivity analysis to explore whether we should consider extended events in addition, as users with triage permission can already make some minor decisions (e.g., decide which issue is labeled as a bug). We discuss this in Section 4.3.

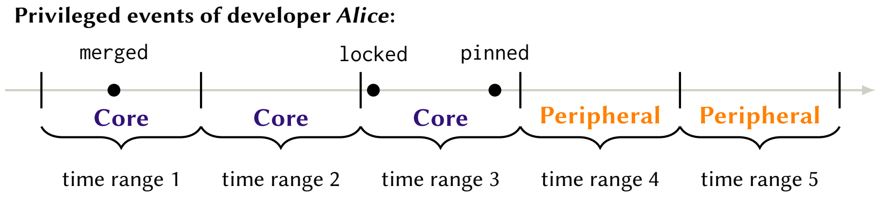

Fig. 2. Analyzing the issue events of developer *Alice*: For each time range, check whether *Alice* has triggered privileged events. If so, *Alice* is considered core (time ranges 1 and 3). For the remaining ranges, check whether *Alice* has triggered privileged events in the directly preceding *and* in the directly succeeding time range. If so, then also consider *Alice* core (time range 2), otherwise peripheral (time ranges 4 and 5).

It is important to note that we cannot know from the data when exactly a developer has received write permission, nor whether or when the write permission may have been revoked. Instead, we only see when the developer happened to trigger an event that requires write permission. So, when considering a developer as core based on the triggered privileged events in a certain time range, we might potentially overlook core developers who only rarely trigger privileged events. To mitigate this risk, we explore how regularly core developers make use of privileged events. This way, we test whether time ranges of 3 to 12 months are long enough—and therefore justified—to identify core developers based on the usage of privileged events within the time range. Therefore, we investigate the following research question:

> **RQ$_1$:** How long is the typical time difference between a developer's events that require, at least, write permission?

As developers might be absent for a particular time range (e.g., due to illness or vacation, etc.), we will exploit a temporal smoothness assumption to improve the accuracy of our method. That is, we treat developers also as core even when they are not triggering privileged events in the current time range but do so in both the previous and succeeding time range (see time range 2 in Figure 2). We call the resulting set of core developers *privileged developers* or D_{priv}, for brevity.

As our approach to identify core developers does not require any manual effort (except for determining which event belongs to which category), it can be performed for each GitHub project fully automatically.

3.3 Validating our Set of Core Developers

After identifying the privileged developers, we need to validate this selection. For this purpose, we ask the following research question:

> **RQ$_2$:** Is the set of privileged developers D_{priv} a sound approximation for the set of core developers?

To answer this question, we need to obtain a reliable set of core developers for a project. Although there are many state-of-the-art classification methods for identifying core developers (as presented in Section 2.3), all these methods might end up in different classifications, making it hard to validate our approach with respect to them. Moreover, as we do not know the accuracy of these methods, we also cannot rely on them for the validation of our approach. Instead, we search for projects that provide publicly available, project-reported maintainer or committer lists, because the developers listed in such lists are (as an official source of information) reliably considered as core developers by the project itself. As projects that collect and provide this information are rare, we can perform this assessment only on a subset of our subject projects. Even if such lists are available, there are different formalisms to publish them. Whereas some projects provide lists with different team member categories in their repositories and keep them up to date (e.g., Node.js provides a steadily updated list of "technical steering committee", "collaborators", and "triagers"[10]), other projects just provide a static list of contributors on external project websites (e.g., as for project Angular[11]) without any historic resolution. Another issue with the latter is to find out who is really a developer and who is just organizational staff, especially if there is a company behind the OSS project. Due to these inconsistencies, we take manual efforts to determine which parts of which lists are relevant for our study. We do not obtain time-resolved lists, but consider only the most recent time range that we analyze for a project. We call the set of developers that have their mandates publicly documented in project-reported lists *documented developers* or D_{doc}.

Note that neither D_{priv} nor D_{doc} are guaranteed to be complete: We cannot be sure that the project-reported lists are regularly updated and really contain all core developers (e.g., some core developers might not want to be listed there), and we can also not be sure that D_{priv} is complete as there could also be core developers that do not have the respective permissions or do not trigger corresponding events. To investigate whether our automatic procedure of extracting D_{priv} is sound, we cross-check it with D_{doc}. For this purpose, we use the *Jaccard Index* as a similarity measure and the measures *Completeness* and *Soundness* as defined as follows:

$$\text{jaccard}(D_{doc}, D_{priv}) = |D_{doc} \cap D_{priv}| \,/\, |D_{doc} \cup D_{priv}| \tag{1}$$

The Jaccard Index lies between 0 and 1, higher values indicating higher similarity.

$$\text{complete}(D_{priv} \mid D_{doc}) = |D_{doc} \cap D_{priv}| \,/\, |D_{doc}| \tag{2}$$

complete($D_{priv} \mid D_{doc}$) is the completeness of D_{priv} with respect to D_{doc}, that is, the proportion of developers in D_{doc} that are also part of D_{priv}.

$$\text{sound}(D_{priv} \mid D_{doc}) = |D_{doc} \cap D_{priv}| \,/\, |D_{priv}| \tag{3}$$

sound($D_{priv} \mid D_{doc}$) is the soundness of D_{priv} with respect to D_{doc}, that is, the proportion of developers in D_{priv} that are also part of D_{doc}.

In addition to validating D_{priv} with respect to D_{doc}, we also need to compare D_{priv} to the outcomes of state-of-the-art classification methods, to obtain a deeper understanding of how our approach performs (see Section 3.5). However, before we do so, let us first introduce the state-of-the-art classification methods that we investigate.

3.4 Developer Classification

Since the set of privileged developers cannot be used to identify potential future maintainers before they get any privileges in the project, and since the necessary data for identifying the set of privileged developers is not necessarily available for projects that use other social coding platforms than GitHub, other classification methods for identifying core developers are still essential. As discussed in Section 2.3, many unsupervised classification methods for identifying core developers do exist. For the network-based methods, there are also various network-construction methods. This yields our main research question:

> **RQ$_3$:** Which metrics and network-construction methods are most accurate in classifying developers into core and peripheral?

[10]https://github.com/nodejs/node/blob/master/README.md#current-project-team-members (accessed: 2022-03-14)
[11]https://angular.io/about?group=Angular (accessed: 2022-03-14)

On the one hand, we investigate established count-based metrics. Particularly, we choose the two most frequently used metrics of this sort (based on our overview of the classification metrics used in the literature in Table 2):
Commit count: the number of commits a developer has made in a certain time period.
LOC count: the number of lines of code (LOC) a developer has changed in a certain time period.
The more central a developer is, that is, the more commits a developer has made or the more LOC the developer has changed, the more likely the developer is a core developer.

On the other hand, to capture the co-coding and co-communication activities of developers, we investigate network-based metrics and different network-construction methods, which we describe in what follows.

Network types & network construction. We investigate three different types of developer networks. All three of them have in common that the vertices represent developers and the edges represent relations among them. However, the different network types differ in the type of relations:

Cochange: Two developers are connected by an edge when they have edited the same file within the same time window [38, 47, 51, 72, 74, 110]. A pure technical view.

Issue: Two developers are connected by an edge when they have contributed to the same issue or pull request (e.g., commenting, reviewing, closing, labeling, etc.) within the same time window [30, 45, 64, 72, 85]. A social view.

Cochange+issue: Contains edges from both above mentioned relations. A socio-technical view.

As also users of a project (not being developers) can contribute comments to issues, we investigate two different cases (i.e., sets of vertices) when constructing issue networks:

All contributors: Construct networks with all users participating in issues, regardless of whether they contributed to the source code.

Code contributors: Only users are considered who have contributed to the source code (i.e., who have authored a commit either in the current or in any previous time range). All other users are removed from the network.

Note that this differentiation only affects the network types *issue* and *issue+cochange*, as *cochange* networks by construction contain only developers who contributed to the source code.

To construct the networks, we split the project data into subsequent time windows of the same length, as common in the literature. We chose to investigate time windows of 3 months [7, 48, 88], 6 months [45, 65, 72], 9 months, and 12 months [79]. Shorter time windows would be threatened by short-term fluctuations (e.g., developers being ill or on vacation). Using larger time windows would neglect project dynamics and developer turnover [37]. As edges that cross time-window boundaries are neglected in such a window-based approach, we additionally investigate sliding windows, that is, instead of using subsequent time windows only, we shift the subsequent window by half the time window, such that subsequent windows overlap and cover the edges which are neglected when not using such a sliding-window approach [49].

When constructing networks, we vary between the following network-construction parameters: A network can either be directed, that is, we consider the order of interactions (e.g., who replies to whom, including transitive relationships as separate edges), or undirected (ignoring the temporal relationship). In addition, we differentiate between simplified networks (only one edge per direction is allowed between one pair of developers, no loops) and unsimplified (multiple edges between one pair of developers, loops are allowed). In Table 4, we provide an overview of all possible choices during network construction as well as the network-based classification metrics that we use, which we explain in the following.

Network-based classification. To classify developers into core and peripheral, we use three established network metrics, which are widely used in the literature (as can be seen in Table 2). In particular, we selected a local centrality metric (degree centrality), a global centrality metric that considers the importance of the developers a developer is interacting with (eigenvector centrality), and a centrality metric that considers the community structure of a network (hierarchy centrality):

Table 4. The network-construction methods and network centrality metrics that we use in our study

Network Type	Network Vertices	Directedness	Simplicity	Time Range	Time Windows	Centrality Metric
cochange	all contributors	directed	simplified		sliding	degree centrality
issue	code contributors	undirected	unsimplified	3 months	subsequent	eigenvector centrality
cochange+issue				6 months		hierarchy centrality
				9 months		
				12 months		

Degree centrality is a local centrality metric. It considers the total degree of a developer in the network, that is, the number of edges a developer has to other developers. The more connections a developer has, the higher its degree centrality [12, 14].

Eigenvector centrality is a global centrality metric. It incorporates the centralities of the developers connected to a developer, to weight the importance of developers by the importance of others they are interacting with. Hence, a developer can have a high eigenvector centrality by either being connected to many other developers or by being connected to developers who also have a high eigenvector centrality [10–12, 14].

Hierarchy centrality considers the community structure of a network. It is calculated by dividing the degree by the local clustering coefficient of a vertex [76]. A high value in hierarchy centrality represents a developer having many connections to other developers, which in turn are loosely connected amongst each other. A low value stands for a developer down in the hierarchy, having only few connections to other developers but which are tightly connected among each other. Core developers should have a high hierarchy centrality due to their coordinative role in the project [48, 49, 51].

All the three network metrics we use have already been used in previous work on developer classification and received a high agreement in the perception of surveyed developers [48].

On each of the metrics, for each time range, we apply the 80% threshold, which is widely used (see Section 2.3). That is, developers whose centrality value is in the upper 20% quantile are considered as core, the remaining ones as peripheral. For data processing, network construction, and centrality computation, we use the library CORONET[12].

3.5 Assessing the Classification Accuracy

To assess the accuracy of the classification methods, we use the set of privileged developers D_{priv} as a point of reference. There are multiple reasons behind this choice. On the one hand, there are practical reasons: As identifying privileged developers is not applicable on projects that use other social coding platforms than GitHub, and since the set of privileged developers cannot be used to identify potential future core developers before they get any privileges, the state-of-the-art classification methods are still necessary. In such cases, it would be helpful to know for practitioners and researchers which of the state-of-the-art methods are closest to our approach on identifying privileged developers. On the other hand, D_{priv} is constructed based on the privileges developers made use of in issue events. Hence, these developers must have received the corresponding permissions in the project from the project's maintainers, which incorporates trustworthy information that comes from a project itself, whereas the established state-of-the-art classification methods do not consider such information.

[12]https://se-sic.github.io/coronet/ (accessed: 2022-03-14)

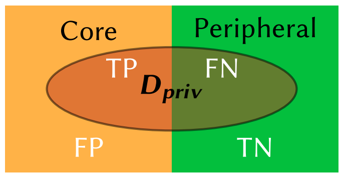

Fig. 3. Visualization of *True Positives (TP)*, *False Positives (FP)*, *True Negatives (TN)*, and *False Negatives (FN)*. The ellipsis represents the set of privileged developers D_{priv}, the two rectangles represent the core/peripheral classification of the used classification method (i.e., the sets of developers that are classified as core or peripheral, respectively).

For assessing the classification accuracy, we compute precision, recall, and F1 score, based on the following notions (see also Figure 3):

True Positives (TP): developers that are classified as core and are part of D_{priv}.
False Positives (FP): developers that are classified as core but are not part of D_{priv}.
True Negatives (TN): developers that are classified as peripheral and are not part of D_{priv}.
False Negatives (FN): developers that are classified as peripheral but are part of D_{priv}. In addition, developers that are part of D_{priv} but that are neither classified as core nor peripheral are also considered to be a false negative (e.g., developers who did never commit but are part of D_{priv} due to triggering privileged issue events are missing in commit-data-based classifications).

Note that the definitions of *FN* and *FP* are specific to our setting: *FN* and *FP* are not defined with respect to D_{priv}, but with respect to the classification. "negative" corresponds to "peripheral"; "positive" corresponds to "core". With that, we compute the following evaluation measures:

$$\text{Precision} = TP \,/\, (TP + FP) \tag{4}$$

$$\text{Recall} = TP \,/\, (TP + FN) \tag{5}$$

$$\text{F1} = 2 \cdot (\text{Precision} \cdot \text{Recall}) \,/\, (\text{Precision} + \text{Recall}) \tag{6}$$

Also note that the choice of the point of reference for the comparison of D_{priv} and the classification results is arbitrary in our case: If we would take the results of the state-of-the-art classification methods as point of reference for assessing the accuracy of D_{priv}, the values of precision and recall would only be swapped (since *FP* would become *FN* then, and vice versa), and, consequently, the F1 score would stay the same. Nevertheless, for the reasons stated above, we chose D_{priv} to be the point of reference in our comparison, to assess the accuracy of the state-of-the-art classification methods.

4 VALIDATION STUDY

In this section, we present the results of our empirical study. We start with an overview of our subject projects, provide the results of our validation of D_{priv}, and report on the assessment of the accuracy of the unsupervised developer classification methods.

4.1 Subject Projects

We investigate 25 highly active software projects hosted on GitHub, covering various project sizes, domains, and numbers of contributors and participants in issues, as the descriptive statistics in Table 5 indicate. We primarily selected very popular projects which are among the most starred GitHub projects in 2020.[13] Most of these projects have also been considered in previous work on GitHub-related research topics. In order not to only analyze

[13]https://www.attosol.com/top-50-projects-on-github-2020/ (accessed: 2020-09-09)

Table 5. Descriptive statistics of our subject projects

Subject Project	Investigated Time Period	# Commit Authors	# Issue Participants	# Commits	# Issues incl. PRs	Project Domain	Programming Languages
Angular	2014-09–2020-09	667	22 859	12 349	38 502	Web dev. platform	TypeScript
Atom	2012-01–2020-12	298	21 047	15 627	21 138	Text editor	JavaScript
Bootstrap	2011-08–2020-12	219	24 744	2 266	31 735	Web front-end framew.	JavaScript, HTML
Deno	2018-05–2020-12	348	3 070	3 417	8 760	Runtime for JavaScript	Rust, JavaScript, TypeScript
DTP	2018-01–2020-04	16	73	633	859	Framew. for data transfer	Java
Electron	2013-05–2020-12	392	15 559	10 664	26 733	Application dev. framew.	C++, TypeScript
Flutter	2015-03–2020-12	683	34 460	13 367	72 504	UI dev. kit	Dart
jQuery	2010-09–2020-12	244	3 118	2 675	4 723	JavaScript library	JavaScript
Keras	2015-03–2019-11	716	12 688	3 471	13 468	Deep learning API	Python
Kubernetes	2014-06–2020-12	2 408	23 220	38 619	97 218	Container management	Go
Moby	2013-01–2020-12	1 154	29 083	14 072	41 731	Softw. containerization	Go
Nextcloud	2016-06–2020-09	355	9 510	9 718	22 689	Cloud server	PHP, JavaScript
Next.js	2016-10–2020-12	867	11 087	3 891	15 344	React framew.	JavaScript, TypeScript
Node.js	2014-11–2020-02	1 793	13 190	12 118	31 372	JavaScript runtime env.	JavaScript, C++, Python
OpenSSL	2013-05–2019-12	400	3 303	8 722	10 639	Crypto library	C, Perl
ownCloud	2012-08–2019-10	393	10 141	18 274	36 178	Cloud server	PHP, JavaScript
React	2013-05–2020-12	796	16 056	6 921	20 252	JavaScript library	JavaScript
Redux	2015-06–2020-12	228	4 123	701	3 931	Container for JavaScript	TypeScript, JavaScript
reveal.js	2011-06–2020-10	141	2 861	1 090	2 762	HTML present. framew.	JavaScript, HTML
TensorFlow	2015-11–2020-12	1 519	35 781	55 499	45 652	Machine learning framew.	C++, Python
three.js	2010-04–2020-12	954	8 280	15 999	20 845	JavaScript library	JavaScript, HTML
TypeScript	2014-07–2020-12	467	18 397	17 934	40 973	JavaScript language	TypeScript
VS Code	2015-11–2020-12	1 001	67 882	49 814	111 073	Integrated dev. env.	TypeScript
Vue	2016-04–2020-11	217	8 754	2 256	9 325	JavaScript UI framew.	JavaScript
webpack	2012-05–2020-12	501	13 091	5 671	11 710	Bundler for modules	JavaScript

popular projects that have a high developer activity, but also cover projects that have a comparably small number of developers, we added project DTP.

4.2 Time Difference between Privileged Events

In RQ$_1$, we ask how long the typical time difference is between a developer's privileged events. In most of the projects, the median time difference is between 0 and 4 days; even the upper quartile of the time differences is below 10 days (except for jQuery, for which it is about 3 weeks). Although there are extreme outliers, for which there are up to 2 000 days between the privileged events of a developer, in most cases, developers make use of privileged events, at least, multiple times a month. As we can see in Figure 4, for more than 88% of the privileged developers the median time difference between privileged events is smaller than 6 months. This holds for all projects except for Vue, for which this time difference is only for 75% of the developers smaller than 6 months, but also reaches 88% at about 7.5 months.

Answer to RQ$_1$: The majority of privileged developers trigger privileged events, at least, once within a couple of weeks. More than 75% of these developers have a median time difference less than 3 months between their privileged events.

As a consequence, determining the set of privileged developers D_{priv} based on privileged events with time windows of 3–12 months is justified. That is, due to applying our temporal smoothness assumption (i.e., also looking at the previous and subsequent range, see Figure 2), our approach is robust to time differences of up

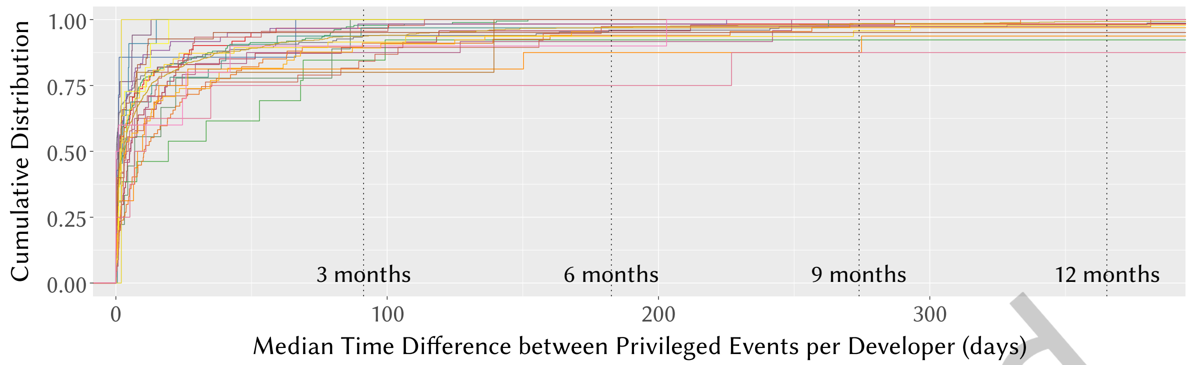

Fig. 4. Cumulative distribution of the median time difference in days between privileged events of a single developer. Each line represents the developers of one subject project.

to three times the window size, ending up in covering core developers who use their permission only once in 9–36 months. As for all but one project more than 88% of the developers using privileged events have a median time difference of less than 6 months between these events, we focus on presenting the results using 6-months ranges in the rest of the paper. For the other time ranges, we refer to our supplementary website.[14]

4.3 Validity of the Set of Privileged Developers D_{priv}

To answer RQ$_2$, we check for the validity of the set of privileged developers D_{priv}. For this purpose, we compare D_{priv} in the most recent time period with project-reported lists (D_{doc}). We were able to obtain D_{doc} for 12 out of our 25 subject projects.

In Table 6, we report the sizes of D_{doc} and D_{priv} for the last-analyzed 6-months time range of each project. When considering only privileged events to detect core developers, we can see that the size of D_{priv} is, in most projects, smaller or nearly equal to the size of the project-reported lists. However, when considering privileged and extended events (see D_{priv+} in Table 6) to detect core developers, in most projects, we are able to extract a much higher number of core developers than reported by the projects' lists. This reinforces our decision that core developers are the developers that have the permission to trigger privileged events. Nevertheless, the similarity of D_{priv} and D_{doc} is highly project-dependent, as the Jaccard Indices between 0.03 and 1.0 indicate. There are two outliers, though: Whereas Kubernetes has way more developers in D_{doc} than in D_{priv}, the list reported by Next.js only contains very few developers. When looking at completeness, we can see that we are able to gather up to 67% of the project-reported developers in D_{priv} (for reveal.js, where only one developer is reported, we reach even 100%). Soundness, on the other hand, reaches up to 94% (for reveal.js even 100%), showing that the vast majority of developers in D_{priv} are also in D_{doc}. We obtain similar results when extracting D_{priv} using other time-window lengths; for more details, we refer to our supplementary website.[14]

Answer to RQ$_2$: The completeness of D_{priv} with respect to D_{doc} seems to be rather small. This is to be expected as we cannot be sure that the project-reported lists are up to date. More importantly, the vast majority of developers in D_{priv} is documented in D_{doc}, which shows that our procedure of extracting D_{priv} is sound.

[14]https://se-sic.github.io/paper-developer-classifications/

Table 6. Validation of the procedure to extract privileged developers for the 12 projects for which we found project-reported lists. (We collected the lists only once per project, temporally close-by to the end of the latest 6-months time range that we analyzed.)

	Angular	Electron	Keras	Kubernetes	Moby	Next.js	Node.js	OpenSSL	React	reveal.js	Vue	webpack
$\lvert D_{doc}\rvert$	30	17	5	238	21	6	108	18	9	1	3	4
$\lvert D_{priv}\rvert$	20	12	5	51	9	25	34	11	10	1	3	6
$\lvert D_{priv+}\rvert$	40	22	10	1 125	9	27	63	199	392	2	3	7
jaccard(D_{doc}, D_{priv})	0.43	0.61	0.43	0.08	0.36	0.03	0.29	0.53	0.46	1.00	0.20	0.25
complete($D_{priv} \mid D_{doc}$)	0.50	0.65	0.60	0.09	0.38	0.17	0.30	0.56	0.67	1.00	0.33	0.50
sound($D_{priv} \mid D_{doc}$)	0.75	0.92	0.60	0.43	0.89	0.04	0.94	0.91	0.60	1.00	0.33	0.33
jaccard(D_{doc}, D_{priv+})	0.49	0.63	0.25	0.08	0.36	0.03	0.50	0.09	0.02	0.50	0.20	0.22
complete($D_{priv+} \mid D_{doc}$)	0.77	0.88	0.60	0.44	0.38	0.17	0.53	0.94	0.78	1.00	0.33	0.50
sound($D_{priv+} \mid D_{doc}$)	0.58	0.68	0.30	0.09	0.89	0.04	0.90	0.09	0.02	0.50	0.33	0.29

D_{doc}: project-reported list,
D_{priv}: core developers based on privileged events,
D_{priv+}: core developers based on privileged+extended events

4.4 Classification-Method Accuracy

Finally, to answer RQ$_3$ about which unsupervised classification methods are most accurate in automatically classifying developers into core and peripheral, we assess their accuracy using precision, recall, and F1. It turned out that there is only a small-to-zero difference between using time windows of 3, 6, 9, or 12 months or using sliding or subsequent windows. Also the difference of the results when using directed or undirected or simplified or unsimplified networks is marginal. For this reason, we present here only the results for subsequent 6-month windows, using unsimplified and directed networks, and briefly put each of the different parts of these results into context; all other results are available on our supplementary website.

In Figure 5, we provide an overview of the accuracy of the different classification methods across all subsequent time ranges of all 25 subject projects. The precision is higher for count-based classifications (median precision between 0.6 and 0.8) and cochange-network-based classifications (median precision around 0.5) than for issue-based (median precision around 0.04) or cochange+issue-based classifications (median precision around 0.04). That is, classifications based on commit data (as in the count-based approaches and on the cochange network) contain a higher percentage of correctly classified core developers over all classified core developers, as compared to the classifications using networks derived from issue data. This may be due to core developers extensively contributing to the source code, whereas in issue networks there are also highly active commenting developers, who appear central in the issue network and are, therefore, wrongly classified as core. In contrast, the recall is higher for classifications based on issue (and issue+cochange) networks (median recall around 1.0) than for commit-data-based classifications (median recall between 0.3 and 0.55). That is, the proportion of developers from D_{priv} that are contained in the classified set of core developers is high, sometimes even close to 100%. This is not unexpected as D_{priv} was derived from issue data. More interestingly, these results indicate that the developers who trigger privileged events also are central in issue networks and may also extensively participate in issues. In general, the number of developers classified as core is higher on issue networks than for commit-data-based classifications, and also higher than in D_{priv}. This coincides with the overall number of developers, which is in issue data way higher than in commit data, which may be a reason for the low precision of the approaches that use issue data.

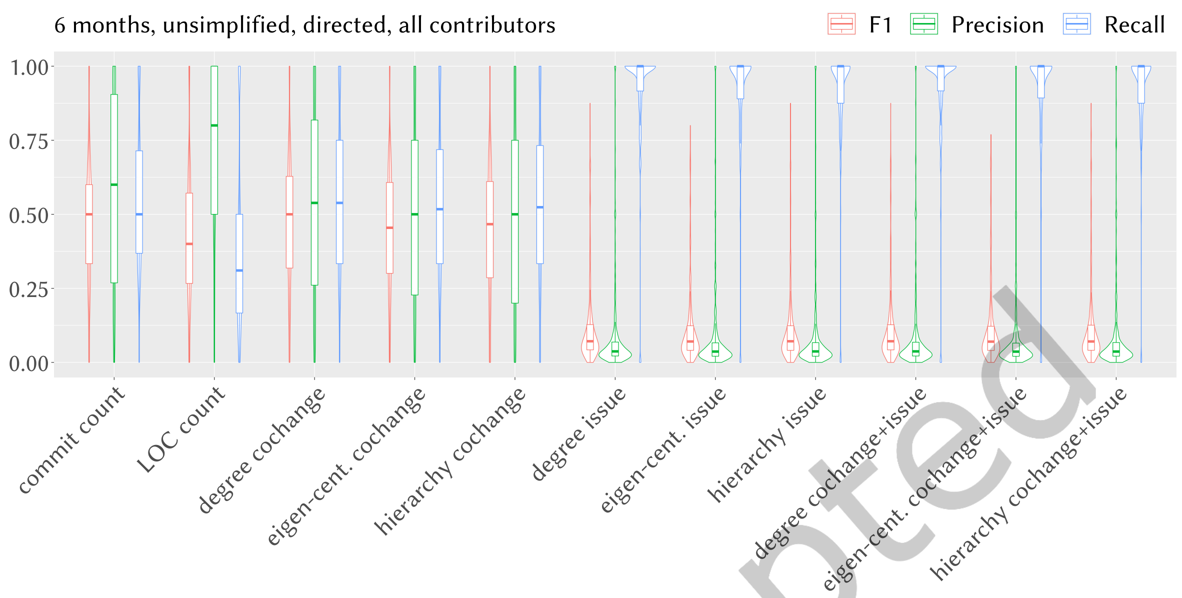

Fig. 5. F1, Precision, and Recall for each classification method with respect to D_{priv}.

It is also worth noting that the recall for methods that rely on commit data is lower than for the remaining methods, as there are developers in D_{priv} that are not part of the commit data and, therefore, neither classified as core nor as peripheral. To account for that, we investigated how many of the developers in D_{priv} are not classified, at all. Across all projects, a median of about 17% of the developers of D_{priv} are not classified by commit-data-based methods. This mainly happens because of two reasons: (1) Core developers may focus on maintenance tasks (such as reviewing and merging) and do not contribute to the source code at all. (2) Some developers only contribute to documentation files (such as README.md), which are not covered in our commit data, as we only keep track of files that contain source code.

To compare the overall performance of the different classification methods, we look at the F1 score. The commit-data-based classification methods have comparably high median F1 scores between 0.4 and 0.5, whereas all the remaining classification methods have much lower median F1 scores of about 0.07. There is almost no difference in the accuracy between issue networks (median F1 around 0.07) and cochange+issue networks (median F1 around 0.07). The reason is that the issue network dominates the cochange+issue network, as there are way more issue discussions than commits (which can also be seen in Table 5). Also the difference in the accuracy of degree centrality, eigenvector centrality, or hierarchy centrality is marginal, which is moreover independent of the network type (mostly degree centrality performs best, in some projects hierarchy centrality performs best). So, in what follows, we neglect the different network metrics and also the cochange+issue networks to focus on comparing count-based, cochange-based, and issue-based classification methods.

To obtain an overview of the overall performance on different projects, we ranked the F1 score (using the median to aggregate across time ranges) of each classification method for each project. The classification methods that resulted in the highest aggregated F1 score on a project received rank 1, the ones that resulted in the lowest aggregated F1 score on a project received rank 11 (as we investigate 11 different classification methods). In Figure 6, we show the distribution of the ranks of the classification methods across all projects. In most projects,

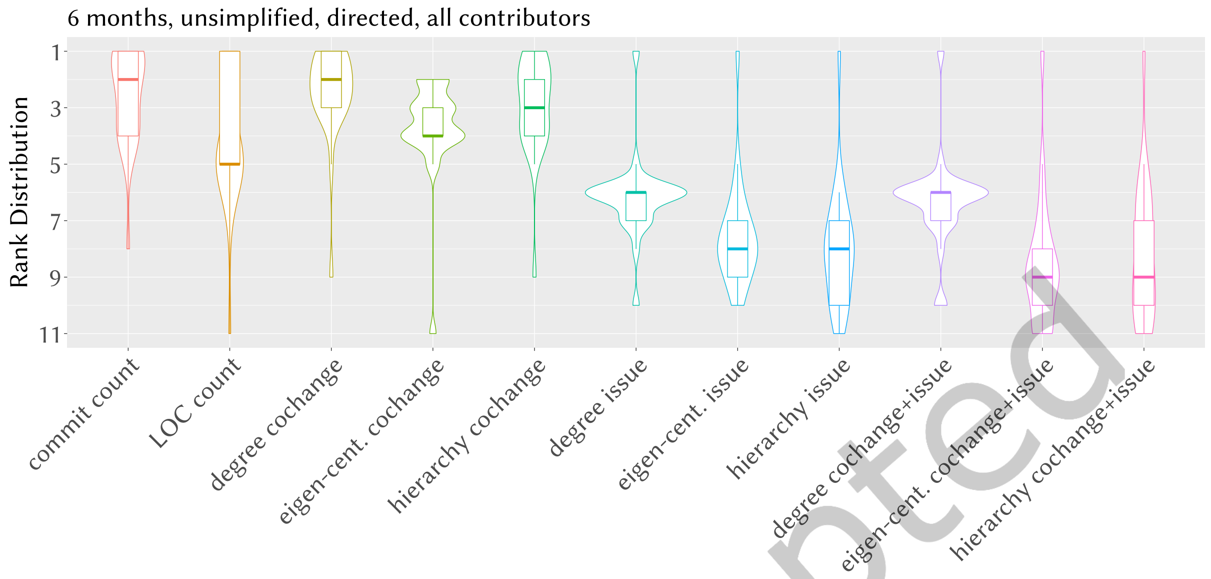

Fig. 6. Distribution of the rank of the different classification methods. That is, for each project, we ranked the median F1 scores of all classification methods. The classification method(s) which had highest median F1 on the project receive rank 1, the method(s) which had the lowest median F1 in the project receive rank 11.

the commit-count-based method or the cochange-based methods perform best. However, there are also projects, in which the issue-based classifications yield higher F1 scores (e.g., projects FLUTTER or DTP; we illustrate accuracies for each project on our supplementary website). These projects have a comparably low number of commits as opposed to a comparably high number of issues.

The overall picture slightly changes when restricting the issue data to code contributors (see Section 3.4). In this case, the issue-based classifications end up with a lower recall (median values around 0.8) but a higher precision (median values around 0.3) and also a higher F1 score (median values around 0.4) than when considering all contributors (see median values stated above). So, when considering only code contributors, the issue-based methods are almost as accurate as the commit-data-based methods, sometimes even more accurate.

Answer to RQ$_3$: In summary, using commit data or cochange networks for classifying core developers performs best. Though, which of these classification methods performs best, is project-dependent. Issue networks often are distorted by users who intensively participate in discussions. Different network-construction methods or time-window lengths do not make a substantial difference for classifying developers into core and peripheral.

5 THREATS TO VALIDITY

The validity of our results may be threatened in various directions (as always in empirical studies). We have grouped the potential threats into four categories that were introduced by Cook and Campbell [26], as suggested by Wohlin et al. [97].

5.1 Construct Validity

Different GitHub projects assign permission levels differently: Some are more liberal, others are more strict. Also individual users use their permissions to a different extent. Thus, relying on the use of privileged events could threaten the validity of computing the set of privileged developers D_{priv}. To mitigate this threat, we investigated two different permission groups (privileged vs. privileged+extended) and four different lengths for the time ranges.

Another threat concerns the way we construct issue networks, as users who do not contribute to the source code may still be part of the issue data. We alleviate this by investigating two cases: When considering only code contributors, the issue-based classification methods perform better than on all contributors, but the overall picture only varies slightly.

Moreover, the network-construction procedure per se may affect our results. To account for that, we investigated different network-construction methods (i.e., directedness, simplicity, time ranges, and time windows, as summarized in Table 4). Our results confirm that the choice of the network-construction method does not make a substantial difference.

5.2 Conclusion Validity

Based on the outcomes related to RQ$_2$, we conclude that D_{priv} is a sound approximation for the set of core developers (that might not be complete, though). This conclusion relies on the reliability of D_{doc} (i.e., project-reported lists). As these project-reported lists are maintained by the project itself, they are an official source of information declaring who is acting as a core developer in a project. Such project-reported lists may be out of date or incomplete (as already discussed in Section 3.3), though. This is why we put on manual efforts to search for these lists and check their relevance for our study. In most cases, when these lists are maintained even within the repository, the commit history shows that they get regularly updated. Still, these lists do not directly map to the time ranges that we used for validating D_{priv}. Since this mapping incorporates substantial manual effort, we performed this validation step only for the last analyzed time range of a subject project. While this might decrease conclusion validity, our manual checks as well as the choice for analyzing the most recent time range ensure that our conclusions are valid, at least, for the time ranges that are close to the last analyzed time ranges, as the project workflows were mostly stable.

Whereas our conclusion that D_{priv} is a sound approximation for the set of core developers is mainly derived from the comparison with D_{doc} (which is an official source of information), the comparison of D_{priv} with the classification outcomes of the state-of-the-art methods (which do not build on officially stated information) provides corroborating evidence.

5.3 Internal Validity

We do not track non-source-code files such as documentation files in our commit data. This is intended, since we aim at identifying core developers who work on the source code. Yet, non-source-code files usually amount only to a small fraction of the files in a software repository.

When we identify developers, we rely on their names and e-mail addresses, to assign them commits and issue events. This could be problematic if developers use different spelling variants for their names or different e-mail addresses (e.g., when they have configured the name or e-mail address in their GitHub account other than in their Git commit configuration). To address this threat, we use the disambiguation heuristic of Oliva et al. [69], which has turned out to be reliable in empirical studies [96], to consider instances that use the same name or the same e-mail address being the same developer, in combination with manual sanity checks. If GitHub users keep their names and e-mail addresses private, we rely on their usernames and on the data that is stored in the commits that are associated with the usernames.

In a similar vein, the detection of bots is a non-trivial task. Therefore, we combine two approaches: We first use the tool BoDeGHa [40] to automatically detect bots based on their commenting behavior in issues, and then we perform manual adjustments in which we also investigate "bot" substrings in usernames and bot marks visible on GitHub. Nevertheless, even if there are bots that we did not detect, this does not threaten our results significantly. The main incentive for detecting and removing bots is to get rid of the bots that are very active and, therefore, distort the network structure or activity counts. If some bots remain in our data that are as active as a usual developer, there is only a low probability that they would distort our results significantly, since we can handle such bots as we handle real developers.

5.4 External Validity

As there are various project shapes, one cannot generalize our results arbitrarily to all OSS projects, at large. Though, we analyzed the complete history of 25 subject projects (starting at the earliest point in time for which all the data sources for a project were available). That is, summing up the investigated time periods of all our 25 projects, we analyzed the data of a combined history of about 160 months, which is a substantial amount of data. Due to the high number of network-construction methods that we investigated (since there are 192 possible combinations of the choices in Table 4), our comprehensive validation study was computationally expensive, which was the reason for limiting the number of subject projects to 25. Albeit, we analyzed 25 projects of different sizes, project domains, and programming languages, which provides already detailed insights.

The results of RQ_2 are potentially threatened by the fact that we were able to obtain the set of documented developers D_{doc} only for 12 out of our 25 subject projects. That is, we validated the set of privileged developers D_{priv} only for these 12 projects with respect to a project-reported list of core developers. We deliberately decided to keep the remaining 13 projects in our study, though, because, this way, we explicitly account for the fact that there are also projects that do not provide such project-reported lists and that, for such projects, it would be beneficial to automatically identify core developers. Nevertheless, using the 12 projects for which we obtained D_{doc} already shows for a multitude of different projects that our approach of identifying core developers based on privileged events is sound and promising. Note that, for answering $RQ_1$and RQ_3, and for conducting the experiment on supervised learning methods (which is part of our future perspective in Section 7), we used all our 25 subject projects to base the corresponding results on a broader dataset. We did not detect any significant differences in the results between the projects for which we were able to obtain D_{doc} and those for which we did not, which also alleviates this threat.

Our method to construct the set of privileged developers D_{priv} is not directly applicable to other platforms than GitHub. Albeit, other platforms might use similar permissions, so the underlying idea of our method should also be transferable.

6 DISCUSSION

As our results for RQ_1 indicate, privileged developers make use of their privileges very often, mostly multiple times within a couple of weeks. Therefore, extracting privileged events within time ranges of several months is sufficient to identify the set of privileged developers. Our results for RQ_2 demonstrate that the set of privileged developers is a sound approximation for the set of core developers, which might not be complete, though, as there can be core developers that do not make use of their privileges or that did not obtain the respective privileges. This is the point at which our approach is stretched to its limits: We cannot identify core developers before they receive privileges. Here, state-of-the-art classifications methods become necessary again, which is why we evaluated their accuracy with respect to our approach, to provide researchers and practitioners the option for selecting an appropriate method for their specific use case. Our results for RQ_3 indicate that *unsupervised* classification methods that use commit data perform slightly better than methods that use issue data. This demonstrates that issue discussions are dominated by users who ask questions or use issues to retrieve knowledge about the project. Using the commit count, which is known to provide only a limited view on organizational structure [48], seems to perform similarly accurate for classifying core developers than various network-based classification methods. This indicates that developers' work on the source code (i.e., having a high coding activity) is, at least, as relevant for becoming a core developer as their interaction with other people. We arrive at a similar conclusion when looking at the used network-construction methods and classification metrics, since they do not make a substantial difference. Sometimes, a more simple, local centrality metric (such as degree centrality) performs even better than eigenvector centrality as a global centrality metric. Nevertheless, these classification metrics perform all pretty similarly, and it is also project dependent as well as data-source dependent which one performs most accurate with respect to the set of privileged developers. The fact that the different methods perform similarly well with respect to our set of privileged developers also indicates that our approach of identifying privileged developers, indeed, is a reasonable approximation for the set of core developers. All in all, instead of recommending a specific method, our main goal is to inform researchers and practitioners about the accuracy of the state-of-the-art unsupervised classification methods.

Our approach provides a viable basis for future research. On the one hand, since receiving a higher level of permissions in a project is also a sign of trust within the community, using the set of privileged developers, we can obtain more information about the characteristics that a developer needs to become a core developer. In particular, we can identify in which time range developers have received their privileges (or, at least, when they have used it for the first time) and search for relevant characteristics of these privileged developers (also considering the role within the network and community structure) before becoming a privileged developer. Such information can be used to improve the search for future candidates to take on more responsibility in the project. In addition, we can also check how the relevant characteristics and the position in the network change after a developer has become part of the set of privileged developers, allowing future researchers to investigate how the activities of a core developer alter after moving up the ladder. This allows for improving community engagement, identification of future project maintainers, and project stability and evolution.

On the other hand, our automatic approach to identify core developers could be used for the development of *supervised* learning methods for identifying core developers. Up until today, to the best of our knowledge, the lack of sufficient volumes of ground-truth data has prevented *supervised* learning methods from being developed. Most of the developer-classification methods used in the literature are either unsupervised or have manually configured thresholds. Hence, project data is only used for evaluation but not for model fitting. Yet, our automatic approach to identify core developers based on privileged issue events could be used for supervised learning methods. To demonstrate the possible benefit of our approach with respect to this research direction, we provide a perspective on supervised classification in the following section by means of a small explorative experiment.

7 A PERSPECTIVE ON SUPERVISED CLASSIFICATION

As already explained in Section 6, state-of-the-art unsupervised developer-classification methods mostly rely on manually configured thresholds or other manual steps. Although we devise an automatic approach to identify core developers based on privileged issue events, it is not capable of identifying potential future core developers before they get any privileges, and it is also not suitable for projects that do not use social coding platforms such as GitHub. Hence, there is still a need for supervised classification methods. Therefore, using our automatic approach to identify core developers based on privileged issue events opens the door for future research to obtain a higher level of maturity when identifying core developers by automated means: Whereas manually configuring the optimal classification thresholds would be a tedious task, supervised learning can be used to automatically learn classification thresholds. Also, combining different classification methods (that use different data and classification metrics) by hand would be impracticable. Instead, learning the weights for combining multiple methods and automatically detecting which combinations are beneficial and which ones not would be possible in a holistic supervised-learning approach. To sketch out the possible benefit of using our automatic core-developer identification approach as a *foundation* for future research in this direction, we conducted a small explorative experiment on how to learn the classification threshold in a binary classification task for a single project or for a set of projects, which we describe and report on in what follows.

Experiment setting: As a preliminary exploration of the potential that supervised learning offers, we pose the problem of role identification as a binary classification task. Instead of using manually configured thresholds, our idea is to learn a function $f : d \rightarrow \{0, 1\}$ that maps a developer d to the core or peripheral class. While some of the classification methods that we used in our validation study are certainly better than others, it is plausible that each of them captures a different dimension of developer roles. For this reason, we construct the input to the classification model to be a combination of all data and metrics used in our validation study. We experimented with several classification models but found that the random forest classifier [22] with a maximum depth of five has the best generalization performance. Due to highly imbalanced classes, we augment the minority class in the training dataset by created synthetic data points using the SMOTE technique [21]. To evaluate the generalization performance of the model, we apply standard cross validation and split the data into separate training and testing sets. We report results in terms of F1 score on the test set for the random forest classifier.

Prediction scenarios: We explore two different prediction scenarios. In the first scenario, we apply k-fold cross validation by splitting the developers randomly into ten groups. We then train ten different models such that each model is trained on nine groups and tested on remaining tenth group. The prediction performance is then averaged over the ten models. This scenario tests how well the model can generalize to a new developer when the model is able to learn from training examples of the roles from every project. For this purpose, we made sure that the test set contains only developers that have not been part of the training set. In the second scenario, we split the developers according to projects so that the training set consists of developers from all projects except the one which forms the test set. In total, 25 models are trained, where each project appears once in the test set and the prediction metrics are averaged. This scenario is more difficult because it tests the model's ability to learn general knowledge about roles in other projects.

Experiment results: For the first scenario, we get an F1 score of 0.73 for our combined model, whereas for the unsupervised methods based on a single metric we get lower F1 scores (0.3–0.5). Similarly, for our second scenario, our combined model (0.62) clearly outperforms each of the unsupervised methods (0.2–0.4). Thus, our explorative small experiment indicates that combining various data sources and socio-technical metrics can help learning promising classification models.

Summary: As our explorative experiment showed, our approach holds the potential to become a foundation for future work on training supervised classification models for the identification of core developers. Consequently, based on the outcomes of this experiment, we formulate two hypotheses that should serve as a starting point for future research on this topic, to sum up our perspective on supervised classification:

Hypothesis 1: Developer classification models that leverage supervised learning are significantly more accurate than state-of-the-art classification methods.

Hypothesis 2: Automatically combining data from different data sources (e.g., commits and issues) and different classification metrics (e.g., the metrics described in Section 3.4) as well as learning appropriate weights for combining multiple methods outperforms state-of-the-art classification methods.

8 CONCLUSION

Identifying core developers in OSS projects is beneficial in various occasions: (1) in large-scale software projects, to improve developer coordination and software quality by revealing who makes decisions and who has consolidated project knowledge; (2) for newcomers and peripheral developers, to get in touch with maintainers and core developers; (3) for companies, to decide whether and how to invest in a project or how to efficiently contribute to an OSS project; (4) and for researchers, to obtain deeper insights into organizational structures of OSS projects and their evolution. Identified core developers that are unknown yet to the project leaders may be candidates to take on more responsibility in the project.

As explicit information on who is core developer is rarely available, we devise an *automatic* method for identifying core developers in GitHub projects based on privileged issue events as well as an assessment of the accuracy of state-of-the-art unsupervised classification methods. Even more than recommending a specific method, we aim at informing researchers and practitioners about the performance of available methods. Our empirical study on 25 GitHub projects reveals that the choice of data source (commit vs. issue data) matters more than the actual classification metric (e.g., the centrality metric), which is a non-obvious result. Our results shall guide practitioners and researchers to choose an appropriate *unsupervised* classification method and provide a solid foundation for future *supervised* learning methods. For this purpose, we have formulated two hypotheses that should serve as guidance for future work.

ACKNOWLEDGMENTS

This work was supported by the German Research Foundation (AP 206/14-1).

REFERENCES

[1] Bram Adams, Ryan Kavanagh, Ahmed E. Hassan, and Daniel M. German. 2016. An Empirical Study of Integration Activities in Distributions of Open Source Software. *Empirical Software Engineering* 21, 3 (2016), 960–1001.

[2] Amritanshu Agrawal, Akond Rahman, Rahul Krishna, Alexander Sobran, and Tim Menzies. 2018. We Don't Need Another Hero? The Impact of "Heroes" on Software Development. In *Proc. Int. Conf. Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. ACM, 245–253.

[3] Ban Al-Ani, Matthew J. Bietz, Yi Wang, Erik Trainer, Benjamin Koehne, Sabrina Marczak, David Redmiles, and Rafael Prikladnicki. 2013. Globally Distributed System Developers: Their Trust Expectations and Processes. In *Proc. Int. Conf. Computer-Supported Cooperative Work (CSCW)*. ACM, 563–574.

[4] Mohamed A. Aljemabi and Zhongjie Wang. 2018. Empirical Study on the Evolution of Developer Social Networks. *IEEE Access* 6 (2018), 51049–51060.

[5] Christian Bird. 2011. Sociotechnical Coordination and Collaboration in Open Source Software. In *Proc. Int. Conf. Software Maintenance (ICSM)*. IEEE, 568–573.

[6] Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz, and Anand Swaminathan. 2006. Mining Email Social Networks. In *Proc. Int. Working Conf. Mining Software Repositories (MSR)*. ACM, 137–143.

[7] Christian Bird, Alex Gourley, Prem Devanbu, Anand Swaminathan, and Greta Hsu. 2007. Open Borders? Immigration in Open Source Projects. In *Proc. Int. Working Conf. Mining Software Repositories (MSR)*. IEEE, 6–6.

[8] Christian Bird, David Pattison, Raissa D'Souza, Vladimir Filkov, and Premkumar Devanbu. 2008. Latent Social Structure in Open Source Projects. In *Proc. Int. Sympos. on Foundations of Software Engineering (FSE)*. ACM, 24–35.

[9] Thomas Bock, Angelika Schmid, and Sven Apel. 2022. Measuring and Modeling Group Dynamics in Open-Source Software Development: A Tensor Decomposition Approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2022), 19:1–19:50.

[10] Phillip Bonacich. 1972. Factoring and Weighting Approaches to Status Scores and Clique Identification. *Journal of Mathematical Sociology* 2, 1 (1972), 113–120.

[11] Phillip Bonacich. 2007. Some Unique Properties of Eigenvector Centrality. *Social Networks* 29, 4 (2007), 555–564.

[12] Stephen P. Borgatti, Martin G. Everett, and Jeffrey C. Johnson. 2018. *Analyzing Social Networks* (2 ed.). Sage.

[13] Amiangshu Bosu and Jeffrey C. Carver. 2014. Impact of Developer Reputation on Code Review Outcomes in OSS Projects: An Empirical Investigation. In *Proc. Int. Sympos. Empirical Software Engineering and Measurement (ESEM)*. ACM, 1–10.

[14] Ulrik Brandes and Thomas Erlebach. 2005. *Network Analysis: Methodological Foundations*. Springer Science & Business Media.

[15] Chris Brown and Chris Parnin. 2019. Sorry to Bother You: Designing Bots for Effective Recommendations. In *Int. Workshop on Bots in Software Engineering (BotSE)*. IEEE, 54–58.
[16] Simon Butler, Jonas Gamalielsson, Björn Lundell, Christoffer Brax, Johan Sjöberg, Anders Mattsson, Tomas Gustavsson, Jonas Feist, and Erik Lönroth. 2021. On Company Contributions to Community Open Source Software Projects. *IEEE Transactions on Software Engineering (TSE)* 47, 7 (2021), 1381–1401.
[17] Fabio Calefato, Filippo Lanubile, and Nicole Novielli. 2017. A Preliminary Analysis on the Effects of Propensity to Trust in Distributed Software Development. In *Proc. Int. Conf. Global Software Engineering (ICGSE)*. IEEE, 56–60.
[18] Edna D. Canedo, Rodrigo Bonifácio, Márcio V. Okimoto, Alexander Serebrenik, Gustavo Pinto, and Eduardo Monteiro. 2020. Work Practices and Perceptions from Women Core Developers in OSS Communities. In *Proc. Int. Sympos. Empirical Software Engineering and Measurement (ESEM)*. ACM, 1–11.
[19] Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2012. Who is Going to Mentor Newcomers in Open Source Projects?. In *Proc. Int. Sympos. on Foundations of Software Engineering (FSE)*. ACM, 1–11.
[20] Marcelo Cataldo and James D. Herbsleb. 2013. Coordination Breakdowns and Their Impact on Development Productivity and Software Failures. *IEEE Transactions on Software Engineering (TSE)* 39, 3 (2013), 343–360.
[21] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. 2002. SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research (JAIR)* 16 (2002), 321–357.
[22] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proc. Int. Conf. Knowledge Discovery and Data Mining (KDD)*. ACM, 785–794.
[23] Can Cheng, Bing Li, Zeng-Yang Li, Yu-Qi Zhao, and Feng-Ling Liao. 2017. Developer Role Evolution in Open Source Software Ecosystem: An Explanatory Study on GNOME. *Journal of Computer Science and Technology (JCST)* 32, 2 (2017), 396–414.
[24] Jailton Coelho, Marco T. Valente, Luciana L. Silva, and André Hora. 2018. Why We Engage in FLOSS: Answers from Core Developers. In *Proc. Int. Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. ACM, 114–121.
[25] Kattiana Constantino, Shurui Zhou, Mauricio Souza, Eduardo Figueiredo, and Christian Kästner. 2020. Understanding Collaborative Software Development: An Interview Study. In *Proc. Int. Conf. Global Software Engineering (ICGSE)*. ACM, 55–65.
[26] Thomas D. Cook and Donald T. Campbell. 1979. *Quasi-experimentation – Design and Analysis Issues for Field Settings*. Houghton Mifflin Company.
[27] Kevin Crowston and James Howison. 2005. The Social Structure of Free and Open Source Software Development. *First Monday* 10, 2 (2005).
[28] Kevin Crowston, Qing Li, Kangning Wei, U. Yeliz Eseryel, and James Howison. 2007. Self-Organization of Teams for Free/Libre Open Source Software Development. *Information and Software Technology (IST)* 49, 6 (2007), 564–575.
[29] Kevin Crowston and Ivan Shamshurin. 2017. Core-Periphery Communication and the Success of Free/Libre Open Source Software Projects. *Journal of Internet Services and Applications (JISA)* 8, 1 (2017), 10:1–10:11.
[30] Kevin Crowston, Kangning Wei, Qing Li, and James Howison. 2006. Core and Periphery in Free/Libre and Open Source Software Team Communications. In *Proc. Hawaii Int. Conf. System Sciences (HICSS)*. IEEE, 118a–118a.
[31] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository. In *Proc. Int. Conf. Computer-Supported Cooperative Work (CSCW)*. ACM, 1277–1286.
[32] Cleidson De Souza, Jon Froehlich, and Paul Dourish. 2005. Seeking the Source: Software Source Code as a Social and Technical Artifact. In *Proc. Int. Conf. Supporting Group Work (GROUP)*. ACM, 197–206.
[33] Luis Felipe Dias, Igor Steinmacher, and Gustavo Pinto. 2018. Who Drives Company-Owned OSS Projects: Internal or External Members? *Journal of the Brazilian Computer Society (JBCS)* 24, 1 (2018), 1–17.
[34] Trung T. Dinh-Trong and James M. Bieman. 2005. The FreeBSD Project: A Replication Case Study of Open Source Development. *IEEE Transactions on Software Engineering (TSE)* 31, 6 (2005), 481–494.
[35] Ikram El Asri, Noureddine Kerzazi, Lamia Benhiba, and Mohammed Janati. 2017. From Periphery to Core: A Temporal Analysis of GitHub Contributors' Collaboration Network. In *Proc. Working Conf. Virtual Enterprises (PRO-VE): Collaboration in a Data-Rich World*. Springer, 217–229.
[36] Mariam El Mezouar, Feng Zhang, and Ying Zou. 2019. An Empirical Study on the Teams Structures in Social Coding using GitHub Projects. *Empirical Software Engineering* 24, 6 (2019), 3790–3823.
[37] Fabio Ferreira, Luciana L. Silva, and Marco T. Valente. 2020. Turnover in Open-Source Projects: The Case of Core Developers. In *Proc. Brazilian Sympos. on Software Engineering (SBES)*. ACM, 447–456.
[38] Harald Gall, Karin Hajek, and Mehdi Jazayeri. 1998. Detection of Logical Coupling based on Product Release History. In *Proc. Int. Conf. Software Maintenance (ICSM)*. IEEE, 190–198.
[39] David Gibson, Jon Kleinberg, and Prabhakar Raghavan. 1998. Inferring Web Communities from Link Topology. In *Proc. Int. Conf. Hypertext and Hypermedia (HT)*. ACM, 225–234.
[40] Mehdi Golzadeh, Alexandre Decan, Damien Legay, and Tom Mens. 2021. A Ground-Truth Dataset and Classification Model for Detecting Bots in GitHub Issue and PR Comments. *Journal of Systems and Software (JSS)* 175 (2021), 110911.

[41] Mehdi Golzadeh, Damien Legay, Alexandre Decan, and Tom Mens. 2020. Bot or Not? Detecting Bots in GitHub Pull Request Activity Based on Comment Similarity. In *Proc. Int. Conf. Software Engineering Workshops (ICSEW)*. ACM, 31–35.

[42] Rajdeep Grewal, Gary L. Lilien, and Girish Mallapragada. 2006. Location, Location, Location: How Network Embeddedness Affects Project Success in Open Source Systems. *Management Science* 52, 7 (2006), 1043–1056.

[43] Rebecca E. Grinter, James D. Herbsleb, and Dewayne E. Perry. 1999. The Geography of Coordination: Dealing with Distance in R&D Work. In *Proc. Int. Conf. Supporting Group Work (GROUP)*. ACM, 306–315.

[44] James D. Herbsleb, Audris Mockus, and Jeffrey A. Roberts. 2006. Collaboration in Software Engineering Projects: A Theory of Coordination. In *Proc. Int. Conf. Information Systems (ICIS)*. Association for Information Systems, 553–568.

[45] Qiaona Hong, Sunghun Kim, Shing Chi Cheung, and Christian Bird. 2011. Understanding a Developer Social Network and its Evolution. In *Proc. Int. Conf. Software Maintenance (ICSM)*. IEEE, 323–332.

[46] Corey Jergensen, Anita Sarma, and Patrick Wagstrom. 2011. The Onion Patch: Migration in Open Source Ecosystems. In *Proc. Europ. Software Engineering Conf. and the Int. Sympos. Foundations of Software Engineering (ESEC/FSE)*. ACM, 70–80.

[47] Andrejs Jermakovics, Alberto Sillitti, and Giancarlo Succi. 2011. Mining and Visualizing Developer Networks from Version Control Systems. In *Proc. Int. Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. ACM, 24–31.

[48] Mitchell Joblin, Sven Apel, Claus Hunsen, and Wolfgang Mauerer. 2017. Classifying Developers into Core and Peripheral: An Empirical Study on Count and Network Metrics. In *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 164–174.

[49] Mitchell Joblin, Sven Apel, and Wolfgang Mauerer. 2017. Evolutionary Trends of Developer Coordination: A Network Approach. *Empirical Software Engineering* 22, 4 (2017), 2050–2094.

[50] Mitchell Joblin, Barbara Eckl-Ganser, Thomas Bock, Angelika Schmid, Janet Siegmund, and Sven Apel. 2023. Hierarchical and Hybrid Organizational Structures in Open-Source Software Projects: A Longitudinal Study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2023). Online first.

[51] Mitchell Joblin, Wolfgang Mauerer, Sven Apel, Janet Siegmund, and Dirk Riehle. 2015. From Developer Networks to Verified Communities: A Fine-Grained Approach. In *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 563–573.

[52] Stefan Koch and Georg Schneider. 2002. Effort, Co-operation and Co-ordination in an Open Source Software Project: GNOME. *Information Systems Journal (ISJ)* 12, 1 (2002), 27–42.

[53] Robert E. Kraut and Lynn A. Streeter. 1995. Coordination in Software Development. *Communications of the ACM* 38, 3 (1995), 69–82.

[54] Rajiv Krishnamurthy, Varghese Jacob, Suresh Radhakrishnan, and Kutsal Dogan. 2016. Peripheral Developer Participation in Open Source Projects: An Empirical Analysis. *ACM Transactions on Management Information Systems (TMIS)* 6, 4 (2016), 1–31.

[55] Irwin Kwan, Adrian Schroter, and Daniela Damian. 2011. Does Socio-Technical Congruence Have an Effect on Software Build Success? A Study of Coordination in a Software Project. *IEEE Transactions on Software Engineering (TSE)* 37, 3 (2011), 307–324.

[56] Amy N. Langville and Carl D. Meyer. 2006. *Google's PageRank and Beyond*. Princeton University Press.

[57] Amanda Lee and Jeffrey C. Carver. 2017. Are One-Time Contributors Different? A Comparison to Core and Periphery Developers in FLOSS Repositories. In *Proc. Int. Sympos. Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–10.

[58] Amanda Lee, Jeffrey C. Carver, and Amiangshu Bosu. 2017. Understanding the Impressions, Motivations, and Barriers of One Time Code Contributors to FLOSS Projects: A Survey. In *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 187–197.

[59] Sherlock A. Licorish and Stephen G. MacDonell. 2013. The True Role of Active Communicators: An Empirical Study of Jazz Core Developers. In *Evaluation and Assessment in Software Engineering (EASE)*. ACM, 228–239.

[60] Sherlock A. Licorish and Stephen G. MacDonell. 2014. Understanding the Attitudes, Knowledge Sharing Behaviors and Task Performance of Core Developers: A Longitudinal Study. *Information and Software Technology (IST)* 56, 12 (2014), 1578–1596.

[61] Ju Long. 2006. Understanding the Role of Core Developers in Open Source Software Development. *Journal of Information, Information Technology, and Organizations (JIITO)* 1, 1 (2006), 75–85.

[62] Yuan Long and Keng Siau. 2007. Social Network Structures in Open Source Software Development Teams. *Journal of Database Management (JDM)* 18, 2 (2007), 25–40.

[63] Luis López-Fernández, Gregorio Robles, Jesus M. Gonzalez-Barahona, and Israel Herraiz. 2006. Applying Social Network Analysis Techniques to Community-Driven Libre Software Projects. *International Journal of Information Technology and Web Engineering (IJITWE)* 1 (2006), 28–50.

[64] Wolfgang Mauerer, Mitchell Joblin, Damian A. Tamburri, Carlos Paradis, Rick Kazman, and Sven Apel. 2022. In Search of Socio-Technical Congruence: A Large-Scale Longitudinal Study. *IEEE Transactions on Software Engineering (TSE)* 48, 8 (2022), 3159–3184.

[65] Andrew Meneely and Laurie Williams. 2011. Socio-technical Developer Networks: Should We Trust Our Measurements?. In *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 281–290.

[66] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. 2002. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11, 3 (2002), 309–346.

[67] João E. Montandon, Marco T. Valente, and Luciana L. Silva. 2021. Mining the Technical Roles of GitHub Users. *Information and Software Technology (IST)* 131 (2021), 106485.

[68] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwen Ye. 2002. Evolution Patterns of Open-Source Software Systems and Communities. In *Proc. Int. Workshop on Principles of Software Evolution (IWPSE)*. ACM, 76–85.

[69] Gustavo A. Oliva, Francisco W. Santana, Kleverton C. M. de Oliveira, Cleidson R. B. de Souza, and Marco A. Gerosa. 2012. Characterizing Key Developers: A Case Study With Apache Ant. In *Proc. Int. Conf. Collaboration and Technology (CRIWG)*. Springer, 97–112.

[70] Gustavo A. Oliva, José Teodoro da Silva, Marco A. Gerosa, Francisco W. S. Santana, Cláudia M. L. Werner, Cleidson R. B. de Souza, and Kleverton C. M. de Oliveira. 2015. Evolving the System's Core: A Case Study on the Identification and Characterization of Key Developers in Apache Ant. *Computing and Informatics* 34, 3 (2015), 678–724.

[71] Fabio Palomba and Damian A. Tamburri. 2021. Predicting the Emergence of Community Smells Using Socio-Technical Metrics: A Machine-Learning Approach. *Journal of Systems and Software (JSS)* 171 (2021), 110847.

[72] Sebastiano Panichella, Gabriele Bavota, Massimiliano Di Penta, Gerardo Canfora, and Giuliano Antoniol. 2014. How Developers' Collaborations Identified from Different Sources Tell us About Code Changes. In *Proc. Int. Conf. Software Maintenance and Evolution (ICSME)*. IEEE, 251–260.

[73] Gustavo Pinto, Igor Steinmacher, and Marco A. Gerosa. 2016. More Common than You Think: An In-Depth Study of Casual Contributors. In *Int. Conf. Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 112–123.

[74] Mathias Pohl and Stephan Diehl. 2008. What Dynamic Network Metrics Can Tell Us About Developer Roles. In *Proc. Int. Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. ACM, 81–84.

[75] Mehvish Rashid, Paul M. Clarke, and Rory V. O'Connor. 2019. A Systematic Examination of Knowledge Loss in Open Source Software Projects. *International Journal of Information Management (IJIM)* 46 (2019), 104–123.

[76] Erzsébet Ravasz and Albert-László Barabási. 2003. Hierarchical Organization in Complex Networks. *Physical Review E* 67, 2 (2003), 026112.

[77] Peter C. Rigby and Ahmed E. Hassan. 2007. What Can OSS Mailing Lists Tell Us? A Preliminary Psychometric Text Analysis of the Apache Developer Mailing List. In *Proc. Int. Working Conf. Mining Software Repositories (MSR)*. IEEE, 23–23.

[78] Peter C. Rigby, Yue Cai Zhu, Samuel M. Donadelli, and Audris Mockus. 2016. Quantifying and Mitigating Turnover-Induced Knowledge Loss: Case Studies of Chrome and a Project at Avaya. In *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 1006–1016.

[79] Gregorio Robles and Jesus M. Gonzalez-Barahona. 2006. Contributor Turnover in Libre Software Projects. In *Int. Conf. Open Source Systems (OSS)*. Springer, 273–286.

[80] Gregorio Robles, Jesus M. Gonzalez-Barahona, and Israel Herraiz. 2009. Evolution of the Core Team of Developers in Libre Software Projects. In *Proc. Int. Working Conf. Mining Software Repositories (MSR)*. IEEE, 167–170.

[81] Pankaj Setia, Balaji Rajagopalan, Vallabh Sambamurthy, and Roger Calantone. 2012. How Peripheral Developers Contribute to Open-Source Software Development. *Information Systems Research* 23, 1 (2012), 144–163.

[82] Pankajeshwara N. Sharma, Bastin T. R. Savarimuthu, and Nigel Stanger. 2017. Boundary Spanners in Open Source Software Development: A Study of Python Email Archives. In *Proc. Asia-Pacific Software Engineering Conf. (APSEC)*. IEEE, 308–317.

[83] Vibha S. Sinha, Senthil Mani, and Saurabh Sinha. 2011. Entering the Circle of Trust: Developer Initiation as Committers in Open-Source Projects. In *Proc. Int. Working Conf. Mining Software Repositories (MSR)*. ACM, 133–142.

[84] Igor Steinmacher, Christoph Treude, and Marco A. Gerosa. 2019. Let Me In: Guidelines for the Successful Onboarding of Newcomers to Open Source Projects. *IEEE Software* 36, 4 (2019), 41–49.

[85] Ashish Sureka, Atul Goyal, and Ayushi Rastogi. 2011. Using Social Network Analysis for Mining Collaboration Data in a Defect Tracking System for Risk and Vulnerability Analysis. In *Proc. India Software Engineering Conf. (ISEC)*. ACM, 195–204.

[86] Damian A. Tamburri, Rick Kazman, and Hamed Fahimi. 2023. On the Relationship Between Organisational Structure Patterns and Architecture in Agile Teams. *IEEE Transactions on Software Engineering (TSE)* 49, 1 (2023), 325–347.

[87] Damian A. Tamburri, Patricia Lago, and Hans van Vliet. 2013. Organizational Social Structures for Software Engineering. *ACM Computing Surveys* 46, 1 (2013), 1–35.

[88] Damian A. Tamburri, Fabio Palomba, Alexander Serebrenik, and Andy Zaidman. 2019. Discovering Community Patterns in Open-Source: A Systematic Approach and its Evaluation. *Empirical Software Engineering* 24, 3 (2019), 1369–1417.

[89] Antonio Terceiro, Luiz Romario Rios, and Christina Chavez. 2010. An Empirical Study on the Structural Complexity Introduced by Core and Peripheral Developers in Free Software Projects. In *Proc. Brazilian Sympos. on Software Engineering (SBES)*. IEEE, 21–29.

[90] Sergio L. Toral, M. Rocío Martínez-Torres, and Federico Barrero. 2010. Analysis of Virtual Communities Supporting OSS Projects Using Social Network Analysis. *Information and Software Technology (IST)* 52, 3 (2010), 296–303.

[91] Marat Valiev, Bogdan Vasilescu, and James D. Herbsleb. 2018. Ecosystem-Level Determinants of Sustained Activity in Open-Source Projects: A Case Study of the PyPI Ecosystem. In *Proc. Europ. Software Engineering Conf. and the Int. Sympos. Foundations of Software Engineering (ESEC/FSE)*. ACM, 644–655.

[92] Zhendong Wang, Yang Feng, Yi Wang, James A. Jones, and David Redmiles. 2020. Unveiling Elite Developers' Activities in Open Source Projects. *IEEE Transactions on Software Engineering (TSE)* 29, 3 (2020), 16:1–16:35.

[93] Kangning Wei, Kevin Crowston, U. Yeliz Eseryel, and Robert Heckman. 2017. Roles and Politeness Behavior in Community-based Free/Libre Open Source Software Development. *Information & Management* 54, 5 (2017), 573–582.

[94] Mairieli Wessel, Bruno Mendes de Souza, Igor Steinmacher, Igor S. Wiese, Ivanilton Polato, Ana P. Chaves, and Marco A. Gerosa. 2018. The Power of Bots: Characterizing and Understanding Bots in OSS Projects. *Proceedings of the ACM on Human-Computer Interaction (HCI)* 2, CSCW (2018), 1–19.

[95] Mairieli Wessel and Igor Steinmacher. 2020. The Inconvenient Side of Software Bots on Pull Requests. In *Proc. Int. Conf. Software Engineering Workshops (ICSEW)*. ACM, 51–55.

[96] Igor S. Wiese, José Teodoro da Silva, Igor Steinmacher, Christoph Treude, and Marco A. Gerosa. 2016. Who is Who in the Mailing List? Comparing Six Disambiguation Heuristics to Identify Multiple Addresses of a Participant. In *Proc. Int. Conf. Software Maintenance and Evolution (ICSME)*. IEEE, 345–355.

[97] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering* (2 ed.). Springer.

[98] Marvin Wyrich and Justus Bogner. 2019. Towards an Autonomous Bot for Automatic Source Code Refactoring. In *Int. Workshop on Bots in Software Engineering (BotSE)*. IEEE, 24–28.

[99] Bo Xu, Donald R. Jones, and Bingjia Shao. 2009. Volunteers' Involvement in Online Community Based Software Development. *Information & Management* 46, 3 (2009), 151–158.

[100] Jin Xu, Yongqin Gao, Scott Christley, and Gregory Madey. 2005. A Topological Analysis of the Open Source Software Development Community. In *Proc. Hawaii Int. Conf. System Sciences (HICSS)*. IEEE, 198a–198a.

[101] Kazuhiro Yamashita, Shane McIntosh, Yasutaka Kamei, Ahmed E. Hassan, and Naoyasu Ubayashi. 2015. Revisiting the Applicability of the Pareto Principle to Core Development Teams in Open Source Software Projects. In *Proc. Int. Workshop on Principles of Software Evolution (IWPSE)*. ACM, 46–55.

[102] Li Yan, Tan Chuan Hoo, and Teo Hock Hai. 2004. The Dynamic Transformation of an Open Source Software Project Leader: A Microorganizational Behavioral Perspective. In *Proc. Pacific Asia Conf. on Information Systems (PACIS)*. Association for Information Systems, 2226–2232.

[103] Yunwen Ye and Kouichi Kishida. 2003. Toward an Understanding of the Motivation of Open Source Software Developers. In *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 419–429.

[104] Liguo Yu. 2008. Self-Organization Process in Open-Source Software: An Empirical Study. *Information and Software Technology (IST)* 50, 5 (2008), 361–374.

[105] Wen Zhang, Ye Yang, and Qing Wang. 2011. Network Analysis of OSS Evolution: An Empirical Study on ArgoUML Project. In *Proc. Int. Workshop on Principles of Software Evolution and ERCIM Workshop on Software Evolution (IWPSE-EVOL)*. ACM, 71–80.

[106] Yuxia Zhang, Minghui Zhou, Audris Mockus, and Zhi Jin. 2019. Companies' Participation in OSS Development – An Empirical Study of OpenStack. *IEEE Transactions on Software Engineering (TSE)* 47, 10 (2019), 2242–2259.

[107] Minghui Zhou, Qingying Chen, Audris Mockus, and Fengguang Wu. 2017. On the Scalability of Linux Kernel Maintainers' Work. In *Proc. Europ. Software Engineering Conf. and the Int. Sympos. Foundations of Software Engineering (ESEC/FSE)*. ACM, 27–37.

[108] Minghui Zhou and Audris Mockus. 2012. What Make Long Term Contributors: Willingness and Opportunity in OSS Community. In *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 518–528.

[109] Minghui Zhou, Audris Mockus, Xiujuan Ma, Lu Zhang, and Hong Mei. 2016. Inflow and Retention in OSS Communities with Commercial Involvement: A Case Study of Three Hybrid Projects. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 2 (2016), 1–29.

[110] Thomas Zimmermann, Andreas Zeller, Peter Weißgerber, and Stephan Diehl. 2005. Mining Version Histories to Guide Software Changes. *IEEE Transactions on Software Engineering (TSE)* 31, 6 (2005), 429–445.