Large-Scale Analysis of the Co-Commit Patterns of the Active Developers in GitHub's Top Repositories

Eldan Cohen
ecohen@mie.utoronto.ca
University of Toronto
Toronto, Canada

Mariano P. Consens
consens@mie.utoronto.ca
University of Toronto
Toronto, Canada

ABSTRACT

GitHub, the largest code hosting site (with 25 million public active repositories and contributions from 6 million active users), provides an unprecedented opportunity to observe the collaboration patterns of software developers. Understanding the patterns behind the social coding phenomena is an active research area where the insights gained can guide the design of better collaboration tools, and can also help to identify and select developer talent. In this paper, we present a large-scale analysis of the co-commit patterns in GitHub. We analyze 10 million commits made by 200 thousand developers to 16 thousand repositories, using 17 of the most popular programming languages over a period of 3 years. Although a large volume of data is included in our study, we pay close attention to the participation criteria for repositories and developers. We select repositories by reputation (based on star ranking), and we introduce the notion of *active developer* in GitHub (observing that a limited subset of developers is responsible for the vast majority of the commits). Using co-authorship networks, we analyze the co-commit patterns of the active developer network for each programming language. We observe that the active developer networks are less connected and more centralized than the general GitHub developer networks, and that the patterns vary significantly among languages. We compare our results to other collaborative environments (Wikipedia and scientific research networks), and we also describe the evolution of the co-commit patterns over time.

ACM Reference Format:
Eldan Cohen and Mariano P. Consens. 2018. Large-Scale Analysis of the Co-Commit Patterns of the Active Developers in GitHub's Top Repositories. In *MSR '18: MSR '18: 15th International Conference on Mining Software Repositories , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3196398.3196436

1 INTRODUCTION

With more than 25 million public repositories,[1] GitHub is the largest online code host. The recent surge in social coding, together with large, publicly available datasets, provides a great opportunity to study the collaboration patterns in large developer networks. Understanding the characteristics of GitHub can help researchers and practitioners design better tools for supporting and enhancing social coding, and perhaps discover new ways to encourage collaboration [32]. Insights gained from analyzing GitHub patterns can also lead to a better assessment of software developer productivity at the individual and group level (as it has been shown for scientific research productivity, e.g. [5]). In fact, many recruiters are relying on the candidates' GitHub profile as a significant factor in hiring decisions [7, 31], and collaborative recruiting platforms (such as `lever.co`, `hikido.com`, `sourcinglab.io`) support identification and selection of applicants based on their social coding activities.

Recent work on mining GitHub, as well as analyzing collaboration patterns in large-scale online environments, highlighted several challenges that should be addressed. Cosentino et al. [6] analyzed 93 research papers that address the task of mining GitHub and found that most papers use datasets of small or medium size, with only 5.4% of the papers applying much-needed longitudinal studies (e.g., evolution analysis). Kalliamvakou et al. [16] explained that while GitHub is a rich source of data on software development, mining GitHub for research purposes should take various potential perils into consideration. Examples for such perils are the large number of repositories that are inactive or have low-activity, the large number of personal repositories, and the limitations of GitHub's API. When studying collaboration patterns in GitHub, other considerations should be taken into account. As a rapidly growing online environment, many of the contributions are extremely small, and many are being rolled-back in a short while. The time dimension is also important, GitHub is a fast-growing network and the collaboration patterns change over time.

The challenges of analyzing large online communities are not unique to social coding sites. Laniado and Tasso [17] used co-authorship networks to analyze collaboration patterns in Wikipedia, adapting to the collaboratively written encyclopedia a central tool in the study of many scientific collaborative environments (e.g., see [2, 23, 24]). They found that traditional co-authorship network methods face challenges to scale to the size of Wikipedia, and without adaptation cannot be applied to the reality of online authoring environments, in which many of the contributions are not sufficient to establish a collaborative relationship. Lack of collaboration can be attributed to the potentially long time gaps between contributions, to the relatively small size of most contributions, or to the fact that a large portion of the edits are being cut-out shortly after. Therefore, the authors propose to limit the relationships in the co-authorship networks only to the main authors of each page. They define a participation criteria that keeps only the main authors, and perform a large-scale analysis of the English Wikipedia

[1] At the end of 2017, as described in https://octoverse.github.com.

MSR '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196436

network. The authors also use temporal co-authorship networks, each corresponding to a time period, allowing them to account for temporal differences and analyze the evolution of the collaboration.

In this work, we present a comprehensive, large-scale analysis of GitHub co-commit patterns, that attempts to address the above mentioned challenges. We consider 10 million commits to 16 thousand repositories, made by 200 thousand developers, over a period of three years. Our study is among those that use the largest amount of GitHub data (only one sixth of the 93 references analyzed in [6] consider more than 100 thousand developers). We select repositories for 17 programming languages, including all the top 10 languages according to IEEE Spectrum ranking,[2] representing diverse characteristics (e.g., imperative/functional, systems/web/scientific, established/recent). Our input data is used to construct 374 temporal co-authorship networks (for multiple languages and time windows), allowing us to analyze co-commit patterns and their evolution over time. Our efforts to analyze a large quantity of GitHub data do not overlook the importance of data quality. We carefully develop participation criteria for both repositories (focusing on the top projects by star-based reputation, and discarding personal and inactive projects), and developers (seeking to eliminate minor committers). Our activity criteria for developers (defining a notion of *active* developers) proves to be very effective; we keep 30% of the developers that are responsible for 90% of the commits.

This is the first work, to our knowledge, to present a comprehensive analysis of the co-commit patterns in GitHub based on co-authorship networks, that addresses the following questions:

RQ1: *What are the co-commit patterns of the active developers, and how do they differ from the patterns of the full set of developers?*

RQ2: *How did the co-commit patterns of the active developers evolve over time, compared to the full set of developers?*

RQ3: *How do the co-commit patterns observed for GitHub compare to the ones observed for scientific collaboration networks and other online collaborative environments such as Wikipedia?*

RQ4: *How do the co-commit patterns differ among programming languages?* Note that RQ4 cross-cuts the previous questions.

A summary answer for each research question appears in page 9, at the end of Section 3 (Results), and following the next section where we describe the design of our study. Section 4 discusses threats to validity, and Section 5 describes related work. We conclude in Section 6, also mentioning future work directions.

We make our dataset and our analysis code and notebooks (including results omitted due to lack of space) available.[3]

2 STUDY DESIGN

2.1 Data Sources and Time Period

We study the co-commit patterns in GitHub, focusing on the communities of the 17 programming languages we select: *C, C#, C++, Clojure, Go, Haskell, Java, Javascript, Julia, OCaml, PHP, Python, R, Ruby, Rust, Scala, Scheme*. We break our analysis based on programming languages to answer *RQ4*. This decision is also justified by the observation (presented in page 8) that the intersection between the programming languages is very small.

[2]http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages

[3]**http://www.cs.toronto.edu/ consens/AnalysisGitHubCoCommit**

Table 1 describes the number of repositories, developers, and commits for each programming languages in our dataset, collected over a period t_c of three years from June 2013 to June 2016. In total, we considered 16,827 unique repositories, 200,205 unique developers, and 9,666,915 unique commits. Notice that these numbers are not the sums of the numbers in Table 1, since some repositories declare more than one language, and some developers contribute to repositories in more than one language. While the repositories for some languages are more active than others (in commits and/or developers), the differences remain within one order of magnitude.

Language	# Repositories	# Developers	# Commits
C	988	24,981	1,687,163
C#	997	14,864	838,687
C++	995	27,049	2,155,219
Clojure	997	3,844	246,797
Go	999	15,215	623,247
Haskell	993	3,886	383,923
Java	996	19,405	1,130,188
Javascript	1,000	39,022	921,785
Julia	993	1,564	154,033
OCaml	967	1,855	298,147
PHP	994	25,388	1,104,364
Python	1,000	36,758	1,233,076
R	987	2,298	217,092
Ruby	998	31,208	1,051,830
Rust	997	4,493	254,694
Scala	996	7,680	446,835
Scheme	930	1,425	207,309

Table 1: Number of repositories, developers, and commits studied during t_c for each language

To account for temporal changes and analyze the evolution of the co-commit patterns to answer *RQ3*, we break t_c into 10 nine-month periods with six-month overlap ($t_{10}, ..., t_1$), with t_1 the most recent three-quarter period from October 2015 to July 2016, t_2 the period from July 2015 to April 2016, etc. We consider nine-month to be a reasonable period to observe collaborative patterns, and we use overlapping time windows with a three-month shift to allow a smoother analysis of the changes observed after each quarter.

2.2 Data Collection and Cleaning

To collect the data, we used GitHub API v3.[4] For each language in our study, we use the *searchRepositories* API call to obtain the top 1,000 repositories based on star ranking (as discussed later, we use star ranking as a popularity measure). Next, we try to take advantage of the *developerStatistics* API call, that provides a weekly summary of commits for each developer. Since this API only provides information for the top 100 developers, we have to resort to mining commits from the commit log of each repository (using *commitLog* API), which is a slower process. Fortunately, only a small number of repositories exceed 100 developers.

The collected dataset is a set of tuples

$$(language, repository, developer, date, commits)$$

[4]https://developer.github.com/v3/

where commits obtained from the developer statistics API have a granularity of a week, while individual commits are obtained from the commit log (since our temporal study uses quarterly units we are not affected by this difference in granularity).

The cleaning process involves filtering out tuples with invalid logins (e.g. an empty login or `"invalid-email-address"` login), which fortunately eliminates only a small portion of the tuples in our dataset (less than 5%) . However, this cleanup is critical to avoid invalid logins being misinterpreted as a developer with a high volume of associated activity.

2.3 Participation Criteria

In this section we present the participation criteria for repositories and developers in our analysis.

Repositories *(reputation criteria).* Our analysis is focused on the co-commit patterns of the top repositories based on reputation. To understand our rationale, this is akin to studying the collaboration patterns of scientific authors by focusing on top tier conferences in each area [4]. We use GitHub's star ranking as a measure of repository reputation. GitHub uses the star ranking in many of its repository rankings including *Trending repositories* [5] and *Explore GitHub* [6] and recommends users to star repositories to allow easy access and to show their appreciation.[7]

We filter out personal repositories (only one committer) as well as inactive repositories. Since some repositories start as personal projects and gradually grow into collaborative projects, while others become inactive over time, we filter personal and inactive repositories for each time period t_i. For instance, for t_c, we filter 2,707 personal and inactive repositories, representing 16% of the repositories. This is a relatively low percentage compared to Kalliamvakou et al.'s finding that 71.6% of the repositories have only one developer (the owner), and that many repositories are inactive [15]. We attribute difference to our reputation criteria (that effectively selects active and collaborative repositories).

Developers *(activity criteria).* A key part of our analysis is the selection of active developers (see *RQ1*). We use $c(d, r)$, the number of commits by a developer d to a specific repository r, and define a parameterized activity threshold based on the average number of commits per repository, denoted as $c_{avg}(r)$. A developer d is considered an *active* developer of repository r iff $c(d, r) > \theta \cdot c_{avg}(r)$ where $\theta \geq 0$ is a parameter that controls the tightness of the activity threshold (for $\theta = 0$, all developers are considered active developers, and for $\theta = 0.75$, only developers that contributed above 75% of the average number of commits are considered active developers).

The purpose of the activity threshold is to get rid of the *long tail* that is associated with activities in social networks. In the context of our GitHub study, it is the long tail of developers that make very few commits for each repository. However, due to the large variety of repositories and commit volumes, we use the mean number of commits per repository. A simple parameter θ adjusts the activity threshold, and we select a θ value that keeps the overall contributions of the removed long tail to approximately 10%.

[5] https://github.com/trending
[6] https://github.com/explore
[7] https://help.github.com/articles/about-stars/

Language	Fraction of Active Developers	Percent of Commits
C	0.226	87.0%
C++	0.240	89.3%
C#	0.287	89.8%
Clojure	0.325	90.3%
Go	0.240	89.6%
Haskell	0.361	90.5%
Java	0.242	88.7%
Javascript	0.177	87.3%
Julia	0.457	88.2%
OCaml	0.370	88.7%
PHP	0.225	88.3%
Python	0.203	87.9%
R	0.365	90.8%
Ruby	0.193	85.9%
Rust	0.326	89.4%
Scala	0.300	88.9%
Scheme	0.371	91.4%
Median	**0.287**	**88.9%**

Table 2: The fraction of active developers and the percent of commits for each language in t_1, for $\theta = 0.75$

We analyzed different θ values in the range $[0.5, 1.25]$, observing the percentage of commits contributed by the *active* developers defined by θ. Table 2 presents a breakdown of the percent of active developers, and the percent of commits made by these developers, for each language in t_1, for our selected parameter value $\theta = 0.75$. We validate that for all languages the active developers contribute approximately 90% of the commits, while the fraction of the active developers ranges between 0.177 and 0.457, with a median of 0.287 (eliminating long tails of developers).

For further validation, this median is very close to 0.33, the median fraction of active developers per repository across all repositories in t_1, as shown in Figure 1 (top), the full histogram of this distribution. As additional information, we also show the full histogram for the fraction of active developers per repository across all repositories in t_c in Figure 1 (bottom), with a median of 0.22.

We also evaluated the option of expanding the repositories included in our study by adding non-top repositories where two or more active developers had collaborated, considering both qualitiative and quantitative aspects. On the qualitative side, using an analogy to the study of scientific authors, we can see that such an expansion could be similar to including non-top publication venues in co-authorship studies. On the quantitiative side, we observe that the number of repositories that would be added to the one thousand top repositories already considered in our study would be small (e.g., for Python just a few hundred repositories would be added, with just low tens involving more than three active developers).

2.4 Analysis Method

To answer the research questions in Section 1, we construct co-authorship networks and analyze the co-commit patterns in GitHub (*RQ1*), studying their evolution (*RQ2*) on per-language basis (*RQ4*), and comparing these results with co-authorship networks of other

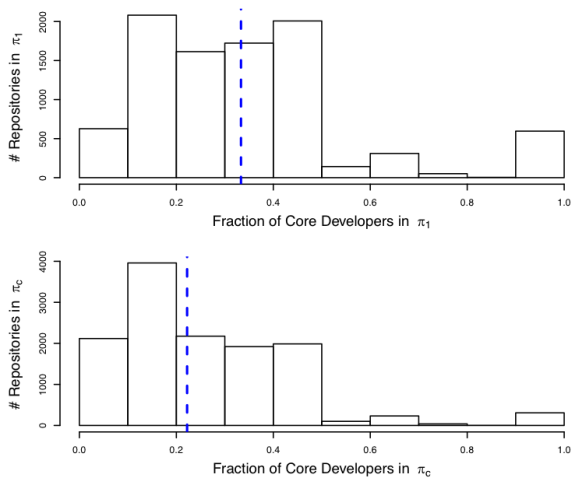

Figure 1: Histogram of the fraction of active developers in the different repositories of π_1 (top) and π_c (bottom)

collaborative environments (*RQ3*). In this section, we describe the process of constructing the networks and the metrics used to analyze the co-commit patterns.

2.4.1 Network Construction. Let D be the set of all developers, R_λ be the set of all repositories in language λ, and $\gamma_i : D \rightarrow 2^R$ be the mapping from a developer to the repositories they commit to during time period t_i. The co-authorship network is an undirected graph $\pi_{i,\lambda} = \langle V, E \rangle$, constructed as follows:

$$V = \{d_j \mid \exists r \in R_\lambda : r \in \gamma_i(d_j)\}$$
$$E = \{(d_j, d_k) \mid \exists r \in R_\lambda : r \in \gamma_i(d_j) \wedge r \in \gamma_i(d_k)\}$$

Thus, each node in $\pi_{i,\lambda}$ is a developer, and each edge connects two developers that contributed commits to the same repository.

We similarly define $\pi^*_{i,\lambda}$, the *core* co-authorship network, by replacing the $\gamma_i : D \rightarrow 2^R$ with $\gamma^*_i : D \rightarrow 2^R$, the mapping from a developer d to the repositories r for which d is an active developer (that is, $c(d,r) > \theta c_{avg}(r)$, as described in page 3). Note that the core is always a subnetwork of the full ($\pi^*_{i,\lambda} \subseteq \pi_{i,\lambda}$).

Our notation uses $\pi_{c,\lambda}$ and $\pi^*_{c,\lambda}$ to denote the full and core networks for t_c (the full three-year period), and omits λ in π_i to refer to the set of networks of all languages (i.e., $\pi_{i,\lambda}$ for all λ). In total we construct and analyze 374 co-authorship networks for the corresponding combinations of 17 programming language × 11 time periods × 2 (core or full).

2.4.2 Network Metrics. The literature on co-authorship networks is large and diverse, and many different characteristics has been proposed (see [23–27]). Due to our interest in the extent and nature of co-commit patterns we focus on five key characteristics of co-authorship networks: connected components, degree distribution, network centralization, community structure, and repository and language overlap.

Connected Components. The *giant component* is the largest connected component in the graph. Typically, the giant component fills a large portion of the graph, while the rest of the nodes are organized in much smaller components [25]. We measure the relative size of the giant component $GC = \frac{|\text{nodes in giant component}|}{|\text{nodes in network}|}$. In our study, we also analyze the second, third and fourth largest components to provide more insights on the structure of the network.

Degree Distribution. Degree is the number of edges connected to a node, and the spread in the degrees is characterized by a distribution $P(k)$ (the probability that a node has exactly k edges) [1]. In our work, we analyze the degree distribution of the network, and compare networks based on the mean and median degree. Due to the different size of the networks, we use the normalized degree of a node n, $ndeg(n) = \frac{100 \cdot degree(n)}{|\text{nodes in network}|}$. Note that *ndeg* is in the range $[0, 100]$, and indicates the percent of the network a node is directly connected to.

Network Centralization. *Betweenness centrality* $B(k)$ [8] measures the centrality of a node k based on the proportion of shortest paths passing through the node, between all pairs of nodes. Therefore, we have that

$$B(k) = \sum_{i,j} \frac{|d_{ikj}|}{|d_{ij}|}$$

where for each pair of nodes i, j, $|d_{ij}|$ is the number of shortest paths between i and j, and $|d_{ikj}|$ is the number of shortest paths that are passing through node k. High betweenness represents a node with more influence and control of communication [8]. As betweenness centrality is measured for individual nodes in the network, *betweenness centralization* B_N is a network-wide metric that measures the relative difference between the most central node and all the other nodes in the network [9]. Hence,

$$B_N = \frac{\sum_i [B(i^*) - B(i)]}{B_S}$$

where i^* is the most central node in the network (i.e., $B(i^*)$ is the highest centrality in the network), and B_S is the betweenness centralization of the star network (the value for the most centralized network). Therefore, higher B_N values characterize networks with a higher level of centrality. In this work, we use B_N as the network centralization metric.

Community Structure. Many networks tend to exhibit a community structure, in which the nodes are organized in tightly-knit groups, between which there are only looser connections [10]. In the context of this work, strong community structure indicates the existence of distinct sub-communities that are mostly co-committing within themselves, and share very few connections between the different sub-communities. *Modularity* is a measure that quantifies the strength of community structure based on the density of connections within each community and between the different communities [27]. Given a partition of the network into K communities, we define a $K \times K$ symmetric matrix e, where e_{ij} is the fraction of all edges in the network that connects community i with j. Modularity M is defined as

$$M = \sum_i (e_{ii} - a_i^2)$$

where a_i denotes the row sum $\sum_j e_{ij}$ (which is the same as the column sum due to symmetry). If the network does not exhibit more dense inter-communities connections than a random network, $M \approx 0$. Values approaching the maximum, $M = 1$, indicate strong community structure. In this work, we use the parallel Louvain method proposed by Staudt and Meyerhenke [29], which uses modularity as its measure of network density.

Language	%GC Core	%GC Full	Ratio
C	22.9%	87.4%	3.82
C#	22.7%	83.5%	3.68
C++	5.5%	88.2%	16.07
Clojure	12.2%	89.6%	7.37
Go	46.9%	95.5%	2.04
Haskell	38.5%	89.6%	2.32
Java	5.1%	89.2%	17.41
Javascript	38.1%	96.2%	2.52
Julia	62.5%	94.8%	1.52
OCaml	43.1%	85.1%	1.98
PHP	30.7%	93.5%	3.05
Python	23.3%	92.6%	3.98
R	18.8%	72.1%	3.84
Ruby	42.3%	96.6%	2.28
Rust	52.4%	97.1%	1.85
Scala	40.8%	84.5%	2.07
Scheme	10.9%	26.5%	2.43
Median	**30.7%**	**89.6%**	**2.52**

Table 3: The relative size of the Giant Component(*GC*) for each language in π_1 and π_1^*

Language	2^{nd} Comp.	3^{rd} Comp.	4^{th} Comp.
C	1.7%	1.4%	1.3%
C#	3.3%	2.3%	2.3%
C++	4.5%	2.6%	1.7%
Clojure	5.7%	3.5%	3.0%
Go	1.6%	1.3%	1.2%
Haskell	2.5%	1.3%	1.2%
Java	5.1%	3.2%	2.7%
Javascript	0.9%	0.8%	0.7%
Julia	0.9%	0.9%	0.6%
OCaml	5.5%	2.9%	2.9%
PHP	3.0%	1.8%	1.6%
Python	11.8%	1.2%	1.1%
R	2.4%	1.5%	1.5%
Ruby	3.2%	2.2%	1.6%
Rust	1.6%	1.6%	0.9%
Scala	1.6%	1.4%	1.3%
Scheme	5.9%	5.0%	4.0%
Median	**3.0%**	**1.6%**	**1.5%**

Table 4: The relative size of 2^{nd}, 3^{rd}, and 4^{th} components in the core sub-network π_1^* for each language

Repository and Language Overlap. In this work, we examine both repository and language overlap. We first analyze the distribution of the number of repositories per user, and the percent of users participating in more than one repository. Then, we analyze the distribution of the number of programming languages per user, and the percent of users that participate in repositories of more than one language (note that language overlap specifically addresses *RQ4* by analyzing the intersection between the communities of the different programming languages).

3 RESULTS

In this section we analyze the co-commit patterns in the constructed co-authorship networks, based on the five chosen metrics. For each metric, we clearly mark the part that addresses each research question. We then summarize the answers to the research questions.

3.1 Connected Components

RQ1: Table 3 shows the relative size of the giant component for each programming language in π_1 and π_1^*, and highlights the difference between them. For most programming languages, the giant component of the full network is above 90%. However, for π_1^* we observe a much smaller giant component. Furthermore, for π_1^*, it varies dramatically between the different programming languages, ranging between 62.5% (Julia) to approximately 5% (Java, C++).

One hypothesis is that the core network includes several relatively large components that if connected, would sum to a dominant giant component. In order to test this hypothesis we measure the relative sizes of the second, third, and fourth largest components. The results, presented in Table 4, contradict this hypothesis, since the second-largest component is quite smaller than the first component. For most languages, the second component is quite smaller than the largest component, and the largest four components do not sum up to the giant component of π_1.

Language	%GC Core	%GC Full	Ratio
C	32.9%	97.2%	2.95
C#	42.1%	94.5%	2.25
C++	7.0%	96.9%	13.86
Clojure	49.7%	98.1%	1.98
Go	67.9%	99.0%	1.46
Haskell	55.6%	96.8%	1.74
Java	20.6%	98.5%	4.77
Javascript	60.3%	99.8%	1.66
Julia	61.7%	95.6%	1.55
OCaml	43.3%	93.0%	2.15
PHP	60.8%	99.1%	1.63
Python	65.8%	99.2%	1.51
R	18.5%	84.8%	4.58
Ruby	75.4%	99.8%	1.32
Rust	64.2%	99.1%	1.54
Scala	48.3%	97.2%	2.01
Scheme	7.0%	34.7%	4.97
Median	**49.7%**	**97.2%**	**1.98**

Table 5: The relative size of the Giant Component(*GC*) for each language in π_c

RQ1/RQ4: These results highlight an important difference between the the different languages that only emerges when we focus on the core network. By filtering out ≈10% of the commits we can see a dramatic decrease in the relative size of the giant component. In the extreme cases (Java and C++), we see a giant component that is more than 16 times smaller.

Table 5 shows the results for the cumulative networks π_c and π_c^*. Naturally, the numbers are a bit higher, as we consider a much larger time period during which components can connect. However,

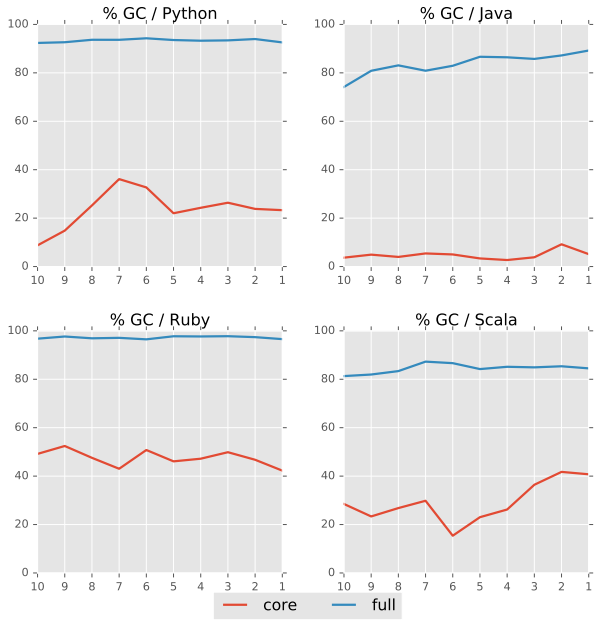

Figure 2: Evolution of the giant component across the temporal networks $\pi_1 - \pi_{10}$

Language	Core avg. *ndeg*	Full avg. *ndeg*	Ratio
C	2.5%	4.1%	1.67
C#	0.6%	0.9%	1.56
C++	0.4%	0.7%	1.72
Clojure	0.5%	1.1%	2.14
Go	0.7%	1.9%	2.54
Haskell	0.7%	2.1%	3.18
Java	0.6%	1.4%	2.22
Javascript	0.4%	0.8%	1.88
Julia	5.2%	21.8%	4.21
OCaml	2.8%	10.1%	3.63
PHP	0.5%	1.0%	2.11
Python	0.5%	0.9%	1.94
R	0.4%	1.3%	2.83
Ruby	1.0%	4.3%	4.39
Rust	1.3%	5.9%	4.64
Scala	1.1%	3.0%	2.59
Scheme	2.0%	6.7%	3.40
Median	**0.7%**	**1.9%**	**2.54**

Table 6: Average *ndeg* for each language in π_1 and π_1^*

the same patterns observed for π_1^* apply for π_c^*. For all languages, the giant component of π_c^* is smaller. Again, when we focus on the core network, we can see clear differences between the different programming languages.

RQ2/RQ4: We examine the evolution of the giant component across the temporal networks $\pi_1 - \pi_{10}$. As the detailed results for all languages cannot fit due to space, we select four languages (Python, Scala, Ruby, Java), that exhibit a range of variations representative of all 17 languages. Figure 2 shows the evolution of the giant component. The relative size of giant component for the core network is constantly smaller than for the full network. The giant component of the full network seems nearly constant and provides little insights on the changing dynamics in the community.

RQ3: In Wikipedia, even when considering the main authors, we see a giant component that is larger than 90% in both the temporal and the cumulative network [17]. Even when considering the topical communities in Wikipedia, the giant component remains approximately 90% [17]. Huang et al. [13] compared the scientific co-authorship networks of Computer science, Maths, Physics, and Biology. These networks demonstrate a large variance in the size of the giant component, ranging from 57.2% to 92.6%.

RQ3/RQ4: More interesting is the difference between the topical communities in computer science. Huang et al. [13] also compared the giant component of six topical communities in computer science, and found that it varies significantly: the largest one was 55.9% (databases) and the smallest one 4.9% (applications). These results are similar to ones observed for the core network in GitHub.

3.2 Degree Distribution

RQ1: Table 6 shows the average normalized degree (*ndeg*) for each language in π_1 and π_1^*. We can see that for all languages, *ndeg* of the core network is smaller than *ndeg* of the full network. We note this is not a direct result of considering a smaller subset of nodes, due to the normalization of the degree. The meaning is that an average *active* developer is connected to a smaller percent of the other *active* developers. To validate the mean is not biased due to some extreme values, we also analyzed the median *ndeg*, and observed the same patterns. We omit these results due to space.

Figure 3 shows the cumulative distribution function (CDF) of the degree distribution for selected 6 languages. For all languages, the core curve is left to the curve of the full network. This supports the pattern observed for the mean and median degree, that the active developers are less connected to the other active developers. Also, the core curve doesn't have the long tail that is often associated with the full network (as can be seen for Python, Javascript and Clojure). Similar patterns have been observed for the cumulative network π_c. We omit the detailed results due to space.

RQ1/RQ4: Note the large difference between the programming languages: the mean *ndeg* values range between 0.4% to 5.2%; and the degree distribution exhibit different patterns.

RQ2/RQ4: Figure 4 shows the evolution of the mean *ndeg* across $\pi_1 - \pi_{10}$. The active developers constantly have lower mean *ndeg*, and the two types of network do not evolve in the exact same manner. We can also see that different languages evolve differently.

RQ3/RQ4: We computed the mean *ndeg* for topical communities in Wikipedia based on the data provided by Laniado and Tasso [17], and for the topical communities in Computer Science based on the data provided by Huang et al. [13]. In Wikipedia, the mean *ndeg* of the examined topical communities ranged between 0.1% to 0.2%. For the topical communities in Computer Science, the value ranged between 0.1% to 0.3%. The result for GitHub is therefore higher. However, this can be attributed to the larger measurement unit of a repository, compared to a scientific paper or a Wikipedia page.

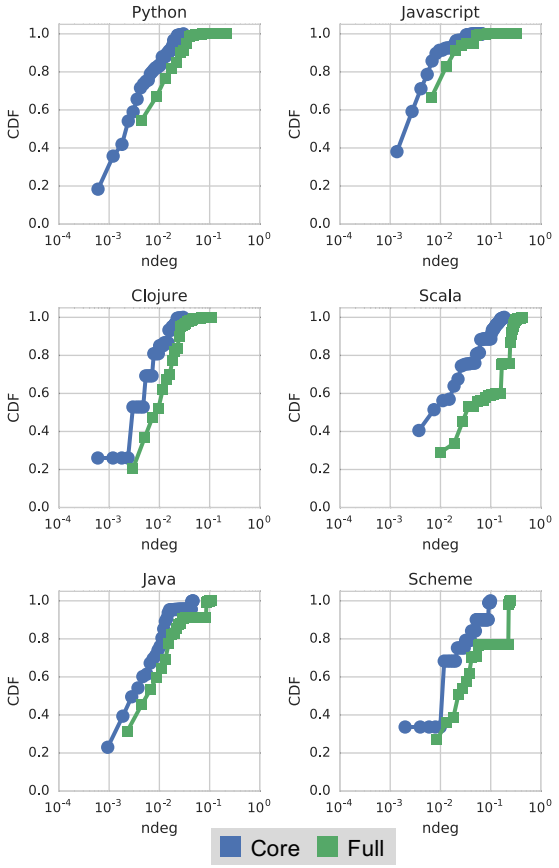

Figure 3: Degree distribution of the core network vs. the full network in π_1

3.3 Network Centralization

RQ1/RQ4: Table 7 shows the *betweenness centralization* (B_N) for the giant component of the core and full network for each language. We focus on the giant component, which represents the largest collaborative component in each language. In all cases but one (Java), the centralization of the core network is much higher than the full, indicating the core networks are more centralized. This observation is consistent with the degree distribution, and suggests the core networks are less collaborative, and the connections between different communities are sparse. We also observe large differences between languages with values ranging from 0.08 to 0.66. We found similar trends in π_c. We omit these results due to space.

RQ2/RQ4: Figure 5 shows the evolution of betweenness centralization across temporal networks $\pi_1 - \pi_{10}$. The plots confirm that the higher centralization of the core remains over time. We note that the centralization of the core is much more sensitive to temporal changes. Java demonstrates particularly high sensitivity, however, it can be attributed to the unique structure of a relatively small giant component, followed by components that are not much smaller.

RQ3/RQ4: Bird et al. [4] measured B_N of 3 Computer Science communities, and find values ranging from 0.0 to 0.6 over time. These are much lower than the core networks, and even lower than most full networks. We could not find similar data for Wikipedia.

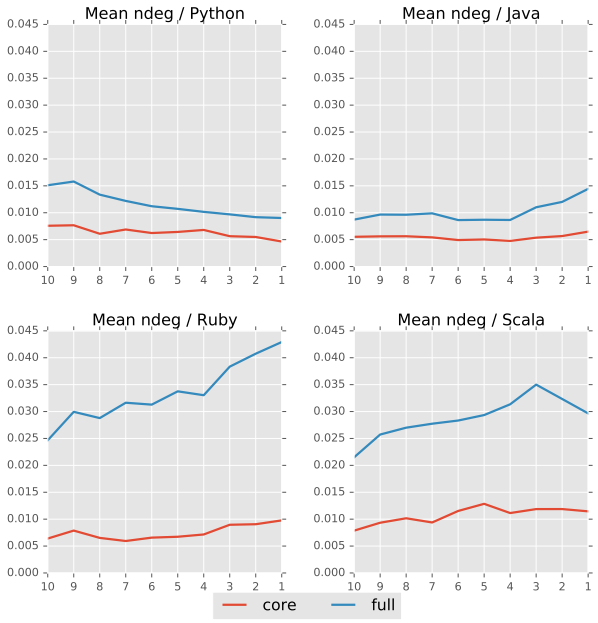

Figure 4: Evolution of the average *ndeg* across the temporal networks $\pi_1 - \pi_{10}$

Language	Core B_N	Full B_N	Ratio
C	0.15	0.05	0.35
C#	0.31	0.14	0.44
C++	0.33	0.20	0.62
Clojure	0.66	0.17	0.25
Go	0.34	0.07	0.20
Haskell	0.33	0.10	0.31
Java	0.08	0.10	1.22
Javascript	0.62	0.30	0.49
Julia	0.13	0.07	0.52
OCaml	0.24	0.08	0.31
PHP	0.32	0.07	0.22
Python	0.50	0.17	0.33
R	0.41	0.20	0.48
Ruby	0.19	0.07	0.37
Rust	0.11	0.06	0.50
Scala	0.40	0.09	0.24
Scheme	0.19	0.13	0.69
Median	**0.32**	**0.10**	**0.37**

Table 7: Betweenness centralization of giant component in π_1 and π_1^*

3.4 Community Structure

RQ1/RQ4: Table 8 shows the community structure and modularity achieved by the parallel Louvain method [29] for each programming languages in π_1 and π_1^* respectively. We can see that, although the core networks are much smaller than the full networks, the number of communities found in the core networks is much higher.

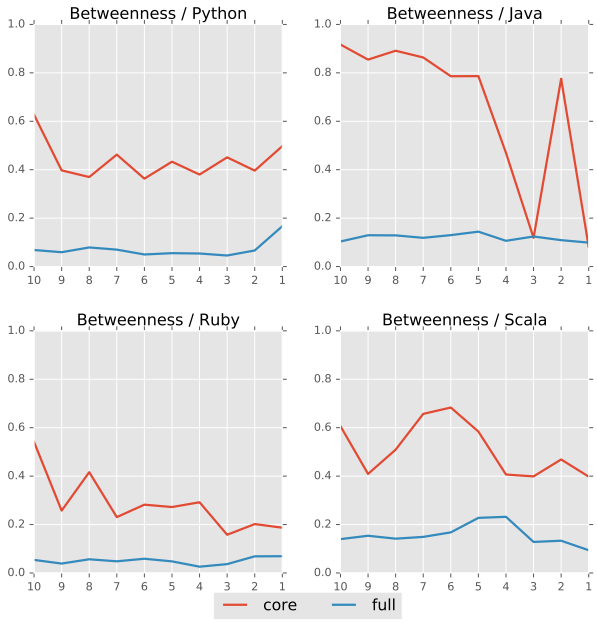

Figure 5: Evolution of the betweenness centralization across the temporal networks $\pi_1 - \pi_{10}$

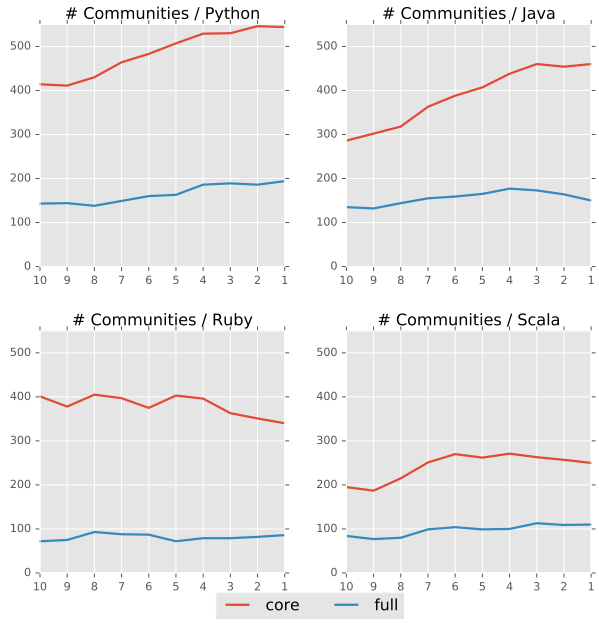

Figure 6: Evolution of the number of communities across the temporal networks $\pi_1 - \pi_{10}$

	Core (π_1^*)			Full (π_1)		
Language	$\#(C)$	$\langle C \rangle$	M	$\#(C)$	$\langle C \rangle$	M
C	516	4.25	0.22	211	45.97	0.20
C#	446	3.39	0.84	206	25.58	0.77
C++	627	4.30	0.95	266	42.28	0.90
Clojure	180	2.24	0.94	55	22.56	0.79
Go	385	4.38	0.83	103	68.30	0.70
Haskell	194	2.68	0.75	54	26.65	0.57
Java	460	3.99	0.89	150	50.47	0.69
Javascript	528	4.79	0.88	130	109.62	0.76
Julia	111	3.07	0.20	37	20.16	0.18
OCaml	93	2.95	0.57	51	14.51	0.45
PHP	437	4.51	0.93	151	58.17	0.83
Python	544	4.83	0.93	194	66.81	0.84
R	184	1.82	0.89	95	9.67	0.70
Ruby	340	5.55	0.78	86	113.88	0.54
Rust	245	2.85	0.49	39	54.82	0.21
Scala	250	3.40	0.64	110	25.69	0.53
Scheme	57	1.77	0.75	44	6.18	0.40
Median	**340**	**3.40**	**0.830**	**103**	**42.28**	**0.686**

Table 8: Community structure for π_1^* and π_1: # of communities $\#(C)$, average community size $\langle C \rangle$, and modularity M.

Naturally, the average size of each core community is much smaller. Also, for all languages, the community structure of the core network has higher modularity compared to the community structure of the full network. These patterns indicate a much stronger community structure of the core networks, and are consistent with previous findings on the co-commit patterns of the active developers. Also, the values vary significantly between the languages. Modularity, as one example, varies from 0.2 to 0.95. Similar patterns have been observed for the cumulative network π_c. We omit the detailed results due to space, and provide the medians across the languages. The median $\#(C)$ is 317.5 for the core networks, compared to 66.5 for the full networks. The median values for $\langle C \rangle$ are 4.5 (core) and 110.5 (full). The median values for M are 0.75 (core) and 0.57 (full). These trends are consistent with our analysis on π_1.

RQ2/RQ4: Figure 6 shows the evolution of the community structure, by following the number of communities across the temporal networks. The number of core communities is much higher, which matches our observation on π_1 and π_c. Furthermore, we see a dramatic increase in the number of core communities for Scala, Java and Python, that is not visible for the full network.

RQ3/RQ4: Bird et al. [4] measured the community structure for several topical communities in Computer Science. Their modularity ranges between 0.634 to 0.916, with a median of 0.824. The median modularity is similar to the one observed for the core networks (0.830). Unfortunately, similar data is not provided for for the topical communities, or for the main authors, in Wikipedia. For the network of all authors, Lizorkin et al. found a modularity of 0.63 [20].

3.5 Repository and Language Overlap

RQ1/RQ4: Table 9 shows the percent of users who are contributing to only one repository in the core and full networks. For nearly all languages, the percent is much higher for the core network, indicating that active developers are much more likely to work on only one repository. We note, however, that the percent varies between the programming languages. Table 10 shows the distribution of the number of programming language used by developers in the core

Language	% in π_1^*	% in π_1
C#	87.1%	83.7%
Scheme	84.2%	73.5%
Rust	74.4%	64.5%
C++	93.6%	89.5%
OCaml	70.4%	71.5%
Javascript	91.8%	84.8%
Julia	59.8%	55.0%
C	86.4%	80.5%
Haskell	73.2%	69.5%
R	82.1%	78.3%
Scala	79.2%	77.1%
Clojure	78.2%	75.5%
PHP	88.8%	83.4%
Go	81.7%	77.5%
Ruby	82.7%	75.5%
Python	91.3%	87.0%
Java	93.5%	89.5%
Median	**82.7%**	**77.5%**

Table 9: Percent of users who are developers in only one repository in the core and in the full networks in π_1

# L	π_1^*	π_1	# L	π_1^*	π_1
1	95.214%	87.983%	6	0.000%	0.020%
2	4.440%	9.907%	7	0.000%	0.006%
3	0.314%	1.664%	8	0.000%	0.007%
4	0.019%	0.320%	9	0.000%	0.004%
5	0.014%	0.085%	10	0.000%	0.001%

Table 10: Distribution of the number of languages used by developers in the core and full networks in π_c

and full networks. Again, active developers are much more likely to only work on repositories in one programming language.

We observed similar patterns of repository overlap for the cumulative network, however we observed interesting exceptions. The more popular languages (Python, Javascript, PHP, etc) exhibit the same patterns, and in most cases it is even strong. For languages with smaller communities (most notable is Julia), we find some cases in which the core community has larger repository overlap, for π_c. The median value for the core networks remains higher. The language overlap in π_c exhibited patterns that are similar to the ones observed for π_1. We omit the detailed results due to space.

RQ2/RQ4: We performed an analysis of the percentage of developers that only contribute to one repository across π_1–π_{10} (omitted due to space). The percentage for the core network is constantly higher compared to the full network, and the difference between the two curves is clear, and is consistent with previous results.

RQ1 Summary: We analyze the co-commit patterns of the *active* developers using five key metrics. Our results indicate that the active developers are less collaborative, more centralized, and have a stronger community structure compared to the population of all GitHub developers.

RQ2 Summary: We study the evolution of the co-commit patterns based on the five metrics, and show that the differences between the active developers and all GitHub developers hold in all time periods in our study. We also show that the core and full networks evolve differently.

RQ3 Summary: We compare our results to similar analyses of Wikipedia and scientific networks in Computer Science (CS). We find some similarities between the component and community structure in GitHub and CS, although GitHub's active developers tend to be more centralized.

RQ4 Summary: We establish that co-commit patterns of active developers varies significantly between the different languages, and they also exhibit different evolution over time. We also observe that differences among languages are comparable to differences among topical communities in CS.

4 THREATS TO VALIDITY

Construct validity. We assume the data collected through GitHub's API is correct and complete, and that the algorithms used to calculate the different metrics [28] are correct with respect to the theoretical metrics. However, we analyzed the data to find inconsistencies (e.g., the discovery and elimination of invalid/empty logins).

Our analysis is exposed to the threat "multiple online personas can cause individuals to be represented as multiple people" present in several papers as described by the survey [12] (as D7).

We tried to validate the collected data with GitHub's Public Data Set[8] and GHTorrent [11], however we could not reproduce the full dataset due to limitations of these sources (GitHub's Public Data Set does not include information on repositories without license,[9] and GHTorrent does not track repositories renames[10]).

Internal validity. We manually picked a set of 17 programming languages that we believed to be diverse and representative of different paradigms. However, we intentionally made sure that we cover the most popular programming language (as indicated earlier, we include all the top 10 ranked languages).

Our study is based on 1000 repositories for each language. While more data can be collected, our choice to use the 1000 most popular repositories based on star ranking indicates that we cover the most visible repositories for each language (which results in a low percent of personal repositories, compared to Kalliamvakou et al.'s findings [16]). An investigation into the starring patterns of GitHub's users is outside the scope of this work, however we adopt GitHub's interpretation of this measure.[11]

In our analysis, we used a chosen θ in our activity threshold for developers. A potential threat is that our analysis is very sensitive to the choice of θ, and a small relaxation of θ might introduce many more active developers and effect the general observed trends. To address this, we repeated our analysis for different values of θ. Table 11 shows the results for key metrics in our analysis (percent of active developers, percent of included commits, size of giant component, and average community size) for t_1 for the different programming languages, when using a relaxed activity threshold with θ=0.5. We

[8] https://cloud.google.com/bigquery/public-data/github
[9] https://github.com/blog/2298-github-data-ready-for-you-to-explore-with-bigquery
[10] http://ghtorrent.org/faq.html
[11] https://help.github.com/articles/about-stars/

Language	%Com	%d	%GC	$\%2^{nd}$	ndeg	$\langle C \rangle$
C#	0.923	0.349	36.1%	3.4%	0.6%	4.51
C++	0.922	0.304	7.7%	5.2%	0.4%	5.72
C	0.905	0.294	34.0%	2.6%	2.7%	6.03
Clojure	0.923	0.405	28.9%	3.2%	0.7%	3.35
Go	0.917	0.322	66.2%	1.1%	0.8%	7.28
Haskell	0.931	0.443	55.3%	2.0%	0.8%	3.94
Java	0.918	0.307	10.2%	9.1%	0.6%	5.71
Javascript	0.897	0.248	50.3%	1.1%	0.4%	7.62
Julia	0.921	0.540	78.4%	0.7%	8.5%	5.68
OCaml	0.918	0.431	48.3%	5.6%	3.7%	3.99
PHP	0.910	0.307	51.8%	3.3%	0.5%	7.16
Python	0.907	0.276	50.8%	1.4%	0.6%	7.57
R	0.940	0.482	30.7%	2.5%	0.7%	2.58
Ruby	0.890	0.275	67.7%	1.6%	1.1%	9.72
Rust	0.920	0.417	66.8%	2.8%	1.5%	4.72
Scala	0.921	0.377	51.0%	1.4%	1.3%	4.78
Scheme	0.940	0.434	12.7%	5.1%	2.5%	2.15
Median θ=0.5	**0.920**	**0.363**	**49.3%**	**2.7%**	**0.8%**	**5.23**
Median θ=0.75	**0.889**	**0.287**	**30.7%**	**3.0%**	**0.7%**	**3.40**
Median θ=0	**1.0**	**1.0**	**89.6%**	**–**	**1.9%**	**42.28**

Table 11: Results for π_1^* with θ=0.5; % commits, % developers, % giant, % 2^{nd} component, % mean *ndeg*, mean community size.

can see that even with a weaker selection criteria, we still filter the majority of developers, while retaining the majority of the commits. The giant component naturally grows, but is still much lower than the full network. The mean degree and the average community size are very similar to the ones observed for θ=0.75. In general, we still observe large changes compared to the full network θ=0 (and large differences among programming languages).

External validity. Our participation criteria for developers is based on the number of commits, which does not take into consideration the content of each commit. More sophisticated participation criteria would require a different dataset (e.g., accessing the code base and automatically analyzing it to evaluate the impact of commits). Since we do not manually or programmatically analyze the content of the 16,827 repositories in our study, we are exposed to the threat of repositories that are not software development projects [16]. However, we expect that our selection of top ranked repositories for each language helps mitigate this threat.

Our analysis is limited to the co-commit patterns in GitHub. Other code hosts exists that could exhibit different patterns. However, with more than 25 million public active repositories, GitHub is the largest code host.

5 RELATED WORK

Co-authorship networks were used to analyze the collaboration patterns in a large variety of scientific collaborative environments [2, 4, 19, 23, 24], and our study adapts this methodology in the context of developer networks.

Few works present a large-scale quantitative analysis of the collaboration patterns in GitHub. Lima et al. analyzed GitHub events, and showed that the distribution of the number of contributors, watchers and followers show a power-law-like shape, that very active users do not necessarily have a large number of followers, and analyzed geographic aspects of collaboration [18]. Thunget al. analyzed the network structure of social coding in GitHub [32]. They modeled the network as a graph and calculated the degree distribution and the shortest path between projects and between developers, and used PageRank to find influental projects and developers. Both [18, 32] partially address *RQ1*, however, they do not cover the five metrics, and do not analyze the evolution of collaboration. Two other major differences with our work are our focus on active developers, and our language-specific analysis.

As described in [6], there are many studies that consider small scale GitHub datasets. Other examples of small scale dataset analyses include Lopez-Fernandez et al. [21] (applying social network analysis to three well-known open source projects), and Bhattacharya et al. [3] (presenting a graph-based model for analysis and prediction of software evolution, studying 11 open source projects).

Previous work also considered social aspects of collaboration in GitHub (e.g., "following"), instead of code authorship collaboration we address. Yu et al. examined the growth mode and follow-network [34]. Jiang et al. analyzed project dissemination [14].

There is also research on developer collaboration outside of GitHub. Xu et al. performed a topological analysis of the open source community based on a SourceForge 2003 data dump [33]. Surian et al. analyzed frequent topological sub-graphs patterns based on a snapshot of SourceForge from 2009 [30]. A recent study [22] that evaluated the validity of social network analysis found that the developer network is supported by developer perceptions.

6 CONCLUSION AND FUTURE WORK

Our large-scale analysis of GitHub co-commit patterns attempts to maximize both the quantity and the quality of the data analyzed, while recognizing the inherent threats and challenges. The resulting study provides answers to research questions including distinguishing a set of active developers, analyzing temporal evolution, comparing developer patterns to ones exhibited by authors of academic papers or Wikipedia articles, and yielding insight into differences among languages. An additional contribution is that the methodology we develop to answer the research questions can be employed to better understand developer productivity and collaboration. One application of this methodology is to analyze smaller teams for the purpose of attracting and retaining developer talent.

There are a number of directions for future work. One is pursuing the analysis of collaborative work within repositories (edges represent commits to the same module, or subsystem), which could produce insights into intra-project collaborative patterns. Another is refining the activity criteria for developers, incorporating additional information (e.g., results from analysis of contributed code).

ACKNOWLEDGMENTS

We thank the reviewers for their valuable feedback. The authors gratefully acknowledge funding from the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] Réka Albert and Albert-László Barabási. 2002. Statistical mechanics of complex networks. *Reviews of Modern Physics* 74, 1 (2002), 47.

[2] Albert-Laszlo Barabâsi, Hawoong Jeong, Zoltan Néda, Erzsebet Ravasz, Andras Schubert, and Tamas Vicsek. 2002. Evolution of the social network of scientific collaborations. *Physica A: Statistical mechanics and its applications* 311, 3 (2002), 590–614.

[3] Pamela Bhattacharya, Marios Iliofotou, Iulian Neamtiu, and Michalis Faloutsos. 2012. Graph-based analysis and prediction for software evolution. In *34th International Conference on Software Engineering (ICSE'12)*. 419–429.

[4] Christian Bird, Premkumar Devanbu, Earl Barr, Vladimir Filkov, Andre Nash, and Zhendong Su. 2009. Structure and dynamics of research collaboration in computer science. In *Proceedings of the 2009 SIAM International Conference on Data Mining (SDM'09)*. 826–837.

[5] Sarvenaz Choobdar, Pedro Ribeiro, Sylwia Bugla, and Fernando Silva. 2012. Comparison of co-authorship networks across scientific fields using motifs. In *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM'12)*. 147–152.

[6] Valerio Cosentino, Javier Luis, and Jordi Cabot. 2016. Findings from GitHub: Methods, datasets and limitations. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR'16)*. 137–141.

[7] Christina DesMarais. 2017. Need Tech Talent? 6 New Places to Look. Retrieved August 24, 2017 from https://www.inc.com/christina-desmarais/6-unexpected-places-to-find-technical-talent.html

[8] Linton C Freeman. 1977. A set of measures of centrality based on betweenness. *Sociometry* (1977), 35–41.

[9] Linton C Freeman. 1978. Centrality in social networks conceptual clarification. *Social networks* 1, 3 (1978), 215–239.

[10] Michelle Girvan and Mark EJ Newman. 2002. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences* 99, 12 (2002), 7821–7826.

[11] Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. IEEE Press, Piscataway, NJ, USA, 233–236. http://dl.acm.org/citation.cfm?id=2487085.2487132

[12] H. Hemmati, S. Nadi, O. Baysal, O. Kononenko, W. Wang, R. Holmes, and M. W. Godfrey. 2013. The MSR Cookbook: Mining a decade of research. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. 343–352. https://doi.org/10.1109/MSR.2013.6624048

[13] Jian Huang, Ziming Zhuang, Jia Li, and C Lee Giles. 2008. Collaboration over time: Characterizing and modeling network evolution. In *Proceedings of the 2008 International Conference on Web Search and Data Mining (WSDM'08)*. 107–116.

[14] J. Jiang, L. Zhang, and L. Li. 2013. Understanding project dissemination on a social coding site. In *2013 20th Working Conference on Reverse Engineering (WCRE'13)*. 132–141. https://doi.org/10.1109/WCRE.2013.6671288

[15] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2014. The promises and perils of mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR'14)*. 92–101.

[16] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2016. An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering* 21, 5 (2016), 2035–2071.

[17] David Laniado and Riccardo Tasso. 2011. Co-authorship 2.0: Patterns of collaboration in Wikipedia. In *Proceedings of the 22nd ACM Conference on Hypertext and Hypermedia (HT'11)*. 201–210.

[18] Antonio Lima, Luca Rossi, and Mirco Musolesi. 2014. Coding Together at Scale: GitHub as a Collaborative Social Network. In *Eighth International AAAI Conference on Weblogs and Social Media (ICWSM'14)*.

[19] Xiaoming Liu, Johan Bollen, Michael L Nelson, and Herbert Van de Sompel. 2005. Co-authorship networks in the digital library research community. *Information Processing & Management* 41, 6 (2005), 1462–1480.

[20] Dmitry Lizorkin, Olena Medelyan, and Maria Grineva. 2009. Analysis of community structure in Wikipedia. In *Proceedings of the 18th International Conference on World Wide Web (WWW'09)*. 1221–1222.

[21] Luis Lopez-Fernandez, Gregorio Robles, Jesus M Gonzalez-Barahona, et al. 2004. Applying social network analysis to the information in CVS repositories. In *International Workshop on Mining Software Repositories (MSR'04)*. 101–105.

[22] A. Meneely and L. Williams. 2011. Socio-technical developer networks: should we trust our measurements?. In *2011 33rd International Conference on Software Engineering (ICSE)*. 281–290. https://doi.org/10.1145/1985793.1985832

[23] Mark EJ Newman. 2001. The structure of scientific collaboration networks. *Proceedings of the National Academy of Sciences* 98, 2 (2001), 404–409.

[24] Mark EJ Newman. 2004. Coauthorship networks and patterns of scientific collaboration. *Proceedings of the National Academy of Sciences* 101, 1 (2004), 5200–5205.

[25] Mark EJ Newman. 2004. Who is the best connected scientist? A study of scientific coauthorship networks. In *Complex networks*. Springer, 337–370.

[26] Mark EJ Newman. 2006. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences* 103, 23 (2006), 8577–8582.

[27] Mark EJ Newman and Michelle Girvan. 2004. Finding and evaluating community structure in networks. *Physical review E* 69, 2 (2004), 026113.

[28] Christian Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. 2014. NetworKit: An Interactive Tool Suite for High-Performance Network Analysis. *CoRR* abs/1403.3005 (2014). http://arxiv.org/abs/1403.3005

[29] Christian L Staudt and Henning Meyerhenke. 2016. Engineering parallel algorithms for community detection in massive networks. *IEEE Transactions on Parallel and Distributed Systems* 27, 1 (2016), 171–184.

[30] Didi Surian, David Lo, and Ee-Peng Lim. 2010. Mining collaboration patterns from a large developer network. In *17th Working Conference on Reverse Engineering (WCRE'10)*. 269–273.

[31] Daniel Terdiman. 2012. Forget LinkedIn: Companies turn to GitHub to find tech talent. Retrieved August 24, 2017 from https://www.cnet.com/news/forget-linkedin-companies-turn-to-github-to-find-tech-talent

[32] Ferdian Thung, Tegawende F Bissyande, David Lo, and Lingxiao Jiang. 2013. Network structure of social coding in GitHub. In *17th European Conference on Software Maintenance and Reengineering (CSMR'13)*. 323–326.

[33] Jin Xu, Yongqin Gao, Scott Christley, and Gregory Madey. 2005. A topological analysis of the open souce software development community. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05)*. 198a–198a.

[34] Yue Yu, Gang Yin, Huaimin Wang, and Tao Wang. 2014. Exploring the Patterns of Social Behavior in GitHub. In *Proceedings of the 1st International Workshop on Crowd-based Software Development Methods and Technologies (CrowdSoft'14)*. 31–36.