How are Issue Units Linked? Empirical Study on the Linking Behavior in GitHub

Lisha Li[1], Zhilei Ren[1], Xiaochen Li[1], Weiqin Zou[2], He Jiang[1]
[1]School of Software, Dalian University of Technology, Dalian, China
[2]State Key Laboratory for Novel Software Technology, Nanjing University, China
{leelisa, li1989}@mail.dlut.edu.cn, {zren, jianghe}@dlut.edu.cn, wqzou@smail.nju.edu.cn

***Abstract*—Issue reports and Pull Requests (PRs) are two important kinds of artifacts of software projects in GitHub. It is common for developers to leave explicit links in issues/PRs that refer to the other issues/PRs during discussions. Existing studies have demonstrated the value of such links in identifying complex bugs and duplicate issue reports. However, there are no broad examinations of why developers leave links within issues/PRs and the potential impact of such links on software development. Without such knowledge, practitioners and researchers may miss various opportunities to develop practical techniques for better solving bug-fixing or feature implementation related tasks. To fill this gap, we conducted the *first* empirical study to explore the characteristics of a large number of links within 642,281 issues/PRs of 16,584 popular (>50 stars) Python projects in GitHub. Specifically, we first constructed an Issue Unit Network (IUN, we refer to issue reports or PRs as issue units) by making use of the links between issue units. Then, we manually checked a sample of 1,384 links in the IUN and concluded six major kinds of linking relationships between issue units. For each kind of linking relationships, we presented some common patterns that developers usually adopted while linking issue units. By further analyzing as many as 423,503 links that match these common patterns, we found several interesting findings which indicate potential research directions in the future, including detecting cross-project duplicate issue reports, using IUN to help better identify influential projects and core issue reports.**

***Index Terms*—issue units, linking Behavior, issues and pull requests, empirical study, software maintenance**

I. Introduction

Issue reports and Pull Requests (PRs) are two important kinds of artifacts of software projects on GitHub [1], [2]. Issue reports are used to record software bugs, required features and any problems encountered in software development and maintenance [3], while PRs are designed to document the code changes and bug fixing patches for software bugs [4]. Till to 2017, more than 100 million issue reports or PRs have been submitted to GitHub [5]. Such a large number of issues/PRs contains a large amount of knowledge about different projects, and thus are commonly referred by developers when they handle newly submitted issues/PRs. Specifically, developers usually leave explicit links (e.g., hyperlinks) in issues/PRs to link relevant issues/PRs together [6].

Existing studies have demonstrated the potential value of such links in identifying complex bugs across projects [6] and detecting duplicate bug reports[1] [7]. However, there are no broad examinations of why developers leave links within issues/PRs and what potential impact such links have on software development. Without such knowledge, practitioners and researchers may miss various opportunities to develop practical techniques that can help developers better perform bug-fixing or feature implementation related tasks. To fill this gap, we conducted the first large-scale study to explore the characteristics of those links developers left in issues/PRs.

Specifically, we first identified 957,132 links from 642,281 issues/PRs from 16,584 Python projects with more than 50 stars in GitHub. For simplicity, in this paper, we refer either issue report or pull request as an issue unit. Based on the links between these issue units, we construct an Issue Unit Network (IUN) to explore the characteristics of developers' linking behavior.

Then, we randomly sampled 1,384 links from the IUN and manually analyzed the relationships of the linked issue units. We found that there are six major kinds of relationships between issue units, namely dependent relationship, duplicate relationship, relevant relationship, referenced relationship, fixed relationship, and enhanced relationship. These relationships well explained for what reasons developers tend to explicitly link issue units. To dig deeper into developers' linking behavior, for each kind of relationships, we further summarized some common patterns which developers followed while linking issue units, e.g., one common pattern *"duplicate of #Num"* for duplicate relationship.

At last, we did a large-scale analysis on 423,503 links that match those common patterns and explored the influence of such links on software development. Some interesting findings can be found in Table I. Specifically, we found that: (1) IUN is an indicator in identifying influential software projects and high-priority issue reports, (2) 7.76% issue units' resolutions heavily depend on the resolutions of other issue units within or across projects, which indicates the importance of developing a mechanism (e.g., by automatically identifying such dependency) to increase developers' awareness of these issue units while resolving relevant problems; (3) 5.97% duplicate relationship happens across projects, which implies a promising direction in detecting duplicates across projects; (4) 54.20% enhanced relationship happens between pull requests. The main reasons to enhance pull requests are the incompleteness of logs, documentation, and test cases. The enhanced relationship between pull requests provides good

[1]Issue reports which concern software bugs are usually called bug reports.

TABLE I: Findings on IUN and their implications.

#	Overview of Linking Behavior	Implications
F1	More than 27% (27.48%) of issue units contain links that refer to other issue units. For those issue units, each issue unit on average has 1.49 links.	Linking issue units is a common practice developers adopted when conducting issue units related tasks.
F2	The in-degree of a project is a potential indicator of the project's influence: The correlation coefficient between a project's in-degree and the number of projects affected by the project is 0.62.	It would be valuable for software platforms like GitHub to identify key activities in high in-degree projects, and then notify any other projects which may be affected by these key activities.
#	**Reasons for Linking Behavior**	**Implications**
F3	There are mainly six kinds of relationships between issue units, including dependent relationship, duplicate relationship, relevant relationship, referenced relationship, fixed relationship and enhanced relationship.	The pervasiveness of these relationships between issue units indicates that it would be worthwhile to develop techniques that can automatically link issue units based on the concluded relationship categories.
F4	7.76% issue units depend on other issue units, which means they can only be fully fixed after the dependent issue units are fixed first.	Some efforts could be made to develop a mechanism that can automatically identify dependent relationship between issue units. Such a mechanism may greatly help developers in fixing issue units of dependent relationships.
F5	40.54% issue units were marked as "duplicate" in more than one day. Notably, 27.96% of issue units were marked as "duplicate" after 10 days. Among those duplicate issue units, 40.97% are duplicate pull requests.	To avoid redundant work in fixing the same problems, it would be valuable to detect both duplicate issue reports and pull requests in practice.
F6	5.97% duplicate relationship happens across projects.	It would be a promising research direction to detect duplicate issue units across projects.
F7	Among referenced links, 63.04% links directed developers to previous comments or discussions that provide additional information for current issue units.	Our results verified that historical issue units indeed contain tremendous knowledge that developers could make use of to address new issue units.
F8	Issue units with high in-degree in IUN tend to have high priority.	Developers could try to make use of the in-degree of the IUN when they need to identify and further study some high-priority or core issue units of a project.
F9	54.20% enhanced relationship happens between PRs. The main reasons to enhance pull requests include the lacking or incompleteness of test scripts,logging statements, etc.	It would be helpful for developers to refer to issue units of enhance relationship, so as to learn how to submit a good pull request.

hints on how to submit high-quality pull requests.

The contributions of this work are as follows:

- To the best of our knowledge, we are the first to broadly examine developers' linking behavior in GitHub.
- We identify six major reasons for why developers link issue units in GitHub projects.
- We propose some major semantic patterns which developer generally followed in linking issue units. These patterns can help to automatically analyze the relationships between issue units.
- We analyze the influence of links between issue units on software development and report several interesting findings which is able to motivate future studies on issue units related tasks.

The remainder of this paper is organized as follows. Section II introduces the background of this study. Section III presents our way to construct the IUN. Section IV summarizes the reasons for linking behavior and introduces the semantic patterns to automatically analyze the reasons. We discuss the influence of the linking behavior on software development in Section V. Threats to validity and related work are presented in Section VI and Section VII respectively. Section VIII concludes this paper.

II. Background

A. Issue Units

Software stakeholders, e.g., developers, testers, managers and users, keep tracking the process of software development and maintenance by documenting software activities. These documents are referred as issue reports [8] and pull requests [2], which are used to report software bugs, suggest new features, explain source code changes, and record bug fixing patches. In this study, we uniformly name issue reports and pull requests as issue units. Due to the importance of issue units in recording software activities, software stakeholders frequently discuss and refer to historical issue units by adding links between issue units [9].

B. Issue Unit Network (IUN)

IUN is a directed acyclic graph to represent the connections between issue units. We denote an IUN graph as IUN(V, E), in which the set of vertexs V contains issue units and the set of edges E refers to links between issue units. For all issue units, if the content of an issue unit v_s contains a link to another issue unit v_t, we create a directed edge from v_s to v_t, denoted as $u_s \rightarrow u_t$. With v_s, v_t and the link $v_s \rightarrow v_t$, an atomic IUN emerges. Repeating the above process, we can eventually create a large IUN regarding all the issue units in a data set. In this IUN, we denote the in-degree of an issue unit v (a vertex) as $\deg_{in}(v)$, which represents the degree of attention that the issue unit u attracts. Similarly, we denote the out-degree of v as $\deg_{out}(v)$, which represents the degree that developers refer to other issue units when discussing v. In this study, we focus on the issue units that $\deg_{in}(v) + \deg_{out}(v) > 0$, namely the non-isolated issue units that connect to another issue unite at least once.

III. Overview of Linking Behavior

In this section, we create IUN to analyze the linking behavior. First, we introduce the data for IUN construction. Then, the process to construct the IUN is presented. At last, we describe the basic information of the constructed IUN.

A. Data Collection

To understand the linking behavior between issue units, in this study, we collect and analyze the issue units from Python projects in GitHub. We analyze these issue units for two reasons. On the one hand, GitHub is currently the largest software development platform in the world [10], which contains millions of software projects. Compared with other repositories, e.g., Bugzilla repository, linking behavior in GitHub is more complex, because developers may frequently add links between different projects [6]. On the other hand, we analyze Python projects since Python is one of the top three most popular programming languages in GitHub [11]. Millions of developers participate in projects written in Python. Hence, analyzing the linking behavior in Python projects may better reflect the activities of the majority of developers in IUN.

We collect the issue units in two steps. First, we collect a set of Python projects in GitHub. We traverse the projects written in Python from 2008 to 2016, since these projects usually obtain enough issue units compared to newly created ones. Of all the collected projects, we investigate the projects with more than 50 stars, because these projects are usually active projects with regular maintenance and public concerns [12]. We implement the above process by GitHub APIs[2]. In this process, 16,584 Python projects are collected. Second, we crawl the issue units from these projects. For each project, we download all the issue units created before Sep. 2017 with GitHub APIs. GitHub APIs return the plain text of the contents of issue units. At last, 2,337,651 issue units are acquired, including 1,205,512 issue reports and 1,132,139 pull requests.

B. IUN Construction

We construct IUN based on the collected issue units. In GitHub, a common way to refer to other issue units is to leave an explicit link in issue units [6], i.e., developers mention an existing issue unit following link patterns like "`#Num`" and "`User/Project#Num`"[3] [13]. For example, in the issue unit `#9243` of the numpy project[4], *Eric-Wieser* added a link `#8916` in the comment "this is a duplicate of `#8916`"[5] to refer to the issue unit `#8916` in the same project. Therefore, we can use patterns of "`#Num`" and "`User/Project#Num`" to match links between issue units. After identifying all the links, we remove the following three types of links:

- Loop links. For an issue unit, developers may add links to the sentences in the same issue unit to quote or comment these sentences. We remove such links, since they do not help developers transfer knowledge between issue units.
- Duplicate links. If links in an issue unit point to the same issue units, we remove the duplicate versions of the links.
- Invalid links. Since the strings matching pattern `#Num` and `User/Project#Num` may be invalid or unreachable links, we ping the identified links to remove invalid ones.

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

[3]`User/Project#Num` refers to the issue unit from https://github.com/User/Project/issues/#Num

[4]https://github.com/numpy/numpy/issues/9243

[5]https://github.com/numpy/numpy/issues/8916

TABLE II: Distribution of links among difference directions

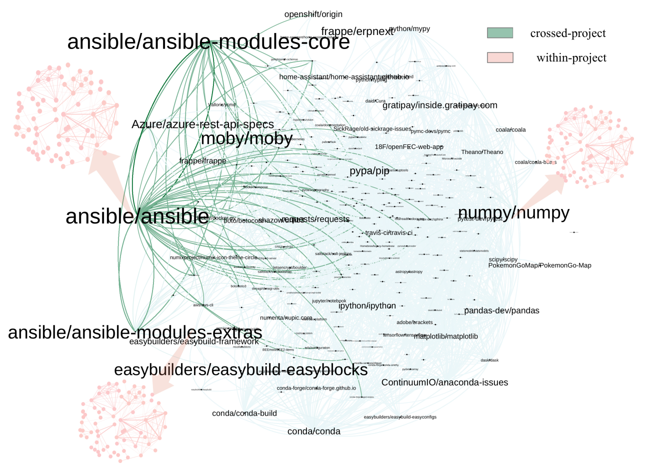
#links	*i*→*i*	*i*→*pr*	*pr*→*i*	*pr*→*pr*	total
within-project	220,435	152,433	258,311	184,943	816,122
cross-project	47,760	17,970	42,743	32,537	141,010
total	268,195	170,403	301,054	217,480	957,132
rating	28.03%	17.8%	31.45%	22.72%	100%

Based on the above rules, we identify 957,132 links from 642,281 issue units. Considering we in total collect 2,337,651 issue units in the high-quality Python projects, the identified 642,281 issue units mean that the linking behavior is a common practice among developers which accounts for nearly one-third (27.48%) issue units. For all issue units contain links, developers on average add 1.49 links in an issue unit.

According to the source and target issue units of a link, we can classify the links into four directions, i.e., *issue report→issue report* (*i*→*i*), *issue report→pull request* (*i*→*pr*), *pull request→issue report* (*pr*→*i*), and *pull request→pull request* (*pr*→*pr*). For example, *i*→*pr* means the source issue unit is an issue report and the target issue unit is a pull request. In addition, on the basis of the project that the issue units belong, we can also classify these links into within-project links and cross-project links. Within-project links link source and target issue units in the same projects. In contrast, cross-project links mean that the source and target issue units belong to different projects. The distribution of links belonging to different directions is shown in Table II.

As show in Table II, the direction of linking behavior is complex. The majority (816,122 links, over 79%) of links are within-project links and about 15% are cross-project links. For the within-project links, a larger proportion of the links is *pr*→*i* (258,311). It seems that developers usually associated a pull request with a specific issue report during software maintenance. For cross-project links, *i*→*i* (47,760) and *pr*→*i* (42,743) happen frequently, which takes up more than 60% of all the cross-project links. It looks like the issue reports for a specific project also attract frequent attention from developers of other projects. Developers discuss these cross-project issues reports in the issue units of their own projects. The above complex behaviors motivate us to specifically analyze the reasons for developers to add links between issue units.

Finding 1. Linking behavior appears in nearly one-third (27.48%) issue units. Among those, each issue unit has 1.49 links on average. These links connect issue units both within and across projects.



🟩 crossed-project
🟥 within-project
openshift/origin
frappe/erpnext
python/mypy
ansible/ansible-modules-core
home-assistant/home-assistant
gratipay/inside.gratipay.com
Azure/azure-rest-api-specs
moby/moby
SickRage/old-sickrage-issues
18F/openFEC-web-app
coala/coala
Theano/Theano
frappe/frappe
pypa/pip
ansible/ansible
requests/requests
numpy/numpy
travis-ci/travis-ci
scipy/scipy
PokemonGoMap/PokemonGo-Map
ipython/ipython
pandas-dev/pandas
numenta/nupic.core
adobe/brackets
matplotlib/matplotlib
ansible/ansible-modules-extras
easybuilders/easybuild-framework
easybuilders/easybuild-easyblocks
ContinuumIO/anaconda-issues
conda-forge/conda-forge.github.io
conda/conda-build
easybuilders/easybuild-easyconfigs
conda/conda

Fig. 1: IUN in terms of projects

C. Basic Information of IUN

In order to investigate the interaction between issue units, we visualize the IUN constructed by the collected Python projects in Fig. 1. Since there are more than 900 thousand links, we present IUN in terms of projects. However, each project still has complex linking behavior as shown in the red subgraphs. In Fig. 1, each node is a project and each edge means a link between issue units in different projects. The size of the project name means the in-degree of a project, which is calculated as the sum of in-degrees of all the issue units in this project. For clarity, we only display projects with more than 100 project in-degrees.

To understand the projects in IUN, we rank all the projects according to the project in-degree. Table III lists the top 20 projects. The first column is the project name. The in-degree, out-degree and the number of stars of each project are listed in the following three columns. In the fifth column, we list the number of projects that has at least one link to the current project. At last, we give the type of each project.

As depicted in Table III, the top 20 projects can be classified into 3 types, including tools for development support, tools for scientific computing, and tools for network and cloud. These high in-degree projects usually affect a large number of other projects. Hence, we can define the in-degree of a project as an indicator to the influence of a project. A large project in-degree means that the issue units in the project have a broader impact on the issue units in other projects, i.e., developers may frequently refer to the issue units in this project when conducting software activities. For example, the issue units in `ansible/ansible` affect more than 2,400 issue units from over 90 other projects. We calculate the Pearson correlation coefficient between the project in-degree of each project in IUN and the number of its affecting projects. The correlation coefficient is 0.62, which shows a position correlation between the project in-degree and affected projects. However, when two projects affect the same number of projects, the project in-degrees may still be different. Hence, we can measure the influence of a project by its project in-degree.

Considering that previous studies usually use project stars to measure the popularity of a project, we find that the project influence is different from project popularity. The correlation coefficient between project in-degree and project star is only 0.17. Hence, these two indicators reflect two aspects of a software project. In addition, identifying the project with high project influence is important. Since the activities in these projects may affect many other projects, it is important for a software platform like GitHub to identify key activities in high in-degree projects, e.g., API deprecation [14], and immediately notice these activities to other projects.

Finding 2. The correlation coefficient between project in-degree and the number of affected projects is 0.62. The project in-degree is one of indicators to evaluate the influence of a project.

IV. Reasons for Linking Behavior

This section discusses the reasons for linking behavior and the way to automatically detect these reasons.

A. Manual Classification Process

To analyze the reasons for linking behavior, we randomly collect a set of links from IUN for analysis. To make our sample representative, we collect the sample by the method of Stratified sample [15] considering a confidence level of 95%, which leads to a sample of 1,384 links. The different strata are represented by four linking directions, including $i \rightarrow pr$, $pr \rightarrow pr$, $pr \rightarrow i$, and $i \rightarrow i$. Each link is associated with two issue units, i.e., a source issue unit and a target issue units.

Based on the sample, we following the strategy used by Ye. et al. [16] to manual analysis on the relationship of two linked issue units. The analysis is performed by two authors of this study, denoted as A1 and A2. They both have computer science background with programming experience over 5 years. The analysis involves three stages. In stage 1, we randomly select 200 links from the sample. A1 analyzed the source and target issue units of the links to infer the reasons for adding the links. A1 wrote the classification rules in classifying these links. Then, A2 used the classification rules to classify the same 200 links, during which the classification rules were revised and refined. In this stage, two authors developed draft categories and classification rules from the 200 links. In stage 2, the authors applied the classification rules in stage 1 to independently classify the remaining 1,184 links into different categories. The authors can either classify a link into the predefined categories or in a new category, if he/she detected a new one. In stage 3, two authors discussed the different cases in the classification. For the 1,184 links, the categories of 137 links are conflicted, i.e., the consistent rate is 0.883. Most conflicts were caused by some new categories detected by any one of the authors in stage 2. In such cases, the definition of the new categories may not be consistent between the two authors. Hence, after classification, two authors conducted a pair-wise discussion to revise and redefine the classification rules and categories to achieve the final ones.

TABLE III: Top 20 projects according to the value of in-degree

project name	in-degree	out-degree	stars	affected project	type
ansible/ansible	2409	2859	27047	94	development support
ansible/ansible-modules-core	1751	1555	1047	40	development support
moby/moby	927	0	47937	136	development support
pytest-dev/pytest	906	176	2293	195	development support
numpy/numpy	890	0	6481	141	scientific computing
pypa/pip	872	362	3992	450	development support
ansible/ansible-modules-extras	838	673	872	27	development support
requests/requests	766	400	31014	307	network and cloud
travis-ci/travis-ci	669	0	6549	407	development support
easybuilders/easybuild-easyblocks	618	0	51	4	development support
ipython/ipython	541	718	12533	142	scientific computing
pandas-dev/pandas	512	578	13419	112	scientific computing
conda/conda	453	692	1993	105	development support
ContinuumIO/anaconda-issues	451	0	215	129	development support
Azure/azure-rest-api-specs	419	0	271	2	network and cloud
shazow/urllib3	417	203	1526	127	network and cloud
matplotlib/matplotlib	417	385	6903	103	scientific computing
tensorflow/tensorflow	415	0	92525	123	scientific computing
frappe/erpnext	390	327	2904	2	scientific computing
conda/conda-build	351	439	113	36	scientific computing

B. Relationship of Linked Issue Units

After manual analysis, we identify six major relationships for the linked issue units, which are the main reasons for the linking behavior.

(1) **Dependent Relationship**. In dependent relationship, developers add a link between two issue units, because the resolution of the source issue unit is dependent on the target one. In other words, the source issue unit is blocked by the target one. Developers usually discuss dependent relationship with phrases such as "*this is blocked by #Num*", "*the upstream issue is #Num*", "*it depends on #Num*", etc. For example, in `ClusterHQ/flocker#1791`, *Itamarst* left an explicit link to explain that "*this (the source issue unit) depends on* `#1767` *(the target issue unit) being merged first.*"

The dependent relationship also exists between cross-project issue units, where the source issue unit is blocked by a problem introduced from another project. Thus, the source issue unit can only be fully fixed after the fixing of the target one. For the cross-project issues, developers usually use a workaround to temporarily handle the source issue unit [6]. The dependent relationship across projects can be deduced by the phrases as "*it's caused by #Num*", "*it was because of #Num*", "*it is introduced by #Num*", etc.

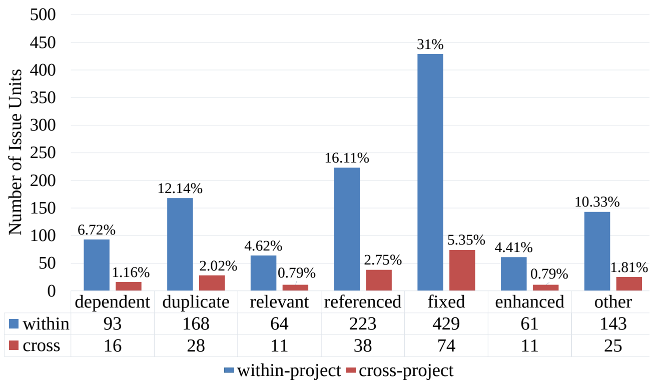
(2) **Duplicate Relationship**. The duplicate relationship means two issue units discuss the same problem. For this relationship, after reading the source issue unit, developers tend to leave an explicit link to its duplicate issue unit. Specifically, in the context of links, developers usually comment the links as "*duplicate of #Num*", "*dupe of #Num*", "*closing as in favor of #Num instead*", etc. These links help other developers look deep into the source and target issue units, and bring more evidences to understand the source issue units.

(3) **Relevant Relationship**. The relevant relationship is different from the duplicate one. When two issue units are duplicate, it means that they address exactly the same problem. However, for the relevant relationship, it usually means that two issue units are similar but not exactly the same. For example, the source and target issue units may discuss similar bugs, share the same topic, or address bugs in the same component or source code file. We find that developers usually explicitly mark a source issue unit as similar or related to another issue unit by phrases like "*related to #Num*", "*this is similar to #Num*", "*they appear to be relevant*".

(4) **Referenced Relationship**. The referenced relationship means the knowledge in the target issue unit may be useful in understanding the source issue unit. In the referenced relationship, the target issue unit may explain a certain concept, approach, or background knowledge for the source issue unit. Besides, the target issue unit may also provide a working example for the source one. For example, in `ansible/ansible-modules-core#2421`, *Gregdek* commented as "*also adding a backport request, per #2557, which means this belong in core review instead. cc: @ansible/core*". The user suggested to refer to the example in the issue unit `ansible/ansible-modules-core#2557` to address the current problem.

(5) **Fixed Relationship**. The fixed relationship means that the source issue unit submits a patch or solution to address the problem reported in the target issue units. Such relationship usually happens in two situations. First, the source issue unit completely solves the reported problem in the target one, thus developers directly close the target issue unit. Second, developers partially solve the reported problem and point out which part of the target issue unit has been solved. The fixed relationship can be easily recognized by some predefined phrases, e.g., "*Fixed by #Num*", "*Done in #Num*", "*#Num addressed this*", and "*Close by #Num*".

(6) **Enhanced Relationship**. The enhanced relationship is defined as that the source issue unit conducts some changes on the target issue unit to make the target issue unit robust or meet the requirements. On the one hand, developers may use the source issue unit to modify or deprecate the contents of the



	dependent	duplicate	relevant	referenced	fixed	enhanced	other
within	93	168	64	223	429	61	143
cross	16	28	11	38	74	11	25

Fig. 2: Distribution of the sampled data.

target issue unit. On the other hand, the target issue unit may be reverted, re-based, or revised in the source issue unit. The enhanced relationship often happens when developers wrongly address the problem in the target issue units. For example, in `DataDog/dd-agent#1274`, *Obi11235* added a new feature that allow users to create custom MySQL metrics for their applications. Because of missing test scripts and logging statements, another two issue units `DataDog/dd-agent#1793` and `DataDog/dd-agent#1673` enhance it.

As shown in Fig. 2, 87.86% links belong to the above six relationships in the 1,384 sampled issue units. Hence, they are the main reasons for the linking behavior between issue units. Besides these relationships, links can also be used to explain a release schedule, indicate the arrangement of software development[6], etc. In this study, we only focus on the identified six relationships for analysis, since they cover more than half of the linking behavior.

Finding 3. We identify six main relationships for linked issue units, including dependent relationship, duplicate relationship, relevant relationship, referenced relationship, fixed relationship and enhanced relationship.

V. Findings in Linking Behavior

In this section, we investigate whether we can automatically identify these relationships by simply applying a series of syntactic patterns and conduct a detail analysis on the linking behavior.

A. Automatic Classification of the Linking Behavior

The syntactic patterns are inspired from the manual classification process. For instance, in `spack/spack#3863`, *Adamjstewart* commented as "*(it is) duplicate of #3346*", in which the keyword "duplicate" indicates a duplicate relationship between issue units `spack/spack#3863` and `spack/spack#3346`. Hence, we take "duplicate" as a syntactic pattern to identify duplicate relationship. Similarly, other relationships have their associated syntactic patterns.

[6]https://github.com/aio-libs/aiohttp/issues/2250

TABLE IV: The number of automatically identified data.

Relationships	Rating	Num		
		all	within-link	cross-link
dependent	7.76%	32,899	25,475	7,424
duplicate	16.69%	70,686	66,339	4,347
relevant	7.9%	33,444	28,747	4,697
referenced	11.53%	48,812	41,855	6,957
fixed	50.85%	215,342	203,324	12,018
enhanced	5.27%	22,320	18,788	3,532

We use a tuple (pattern, pattern_loc) to represent each syntactic pattern. The "pattern" denotes the keywords or regular expression to match the context of a link. The "pattern_loc" is the location of a pattern, which has three values $\{-1, 0, 1\}$. The value "-1" means that we use the pattern to match the context before a link. "1" indicates that the pattern is used to match the context after a link. "0" represents that we match either before or after the link. In this study, we look up five words before or after a link according to our pilot study.

Table V lists all the pattern tuples used in this study. The last column presents the link direction. For example, for the duplicate relationship, we only use these patterns to match links from issue report to issue report and from pull request to pull request, because issue reports and pull requests are never labeled as duplicate in our sample. If a sentence is matched by more than one pattern, the relationship is determined as the one that the closest pattern belongs to.

Given the 957,132 links we collected, the patterns match 423,503 links. Hence, the matching rate is 0.44. The low matching rate causes by the fact that the meaning of a sentence can be expressed in diverse ways, many of which are not exactly the same with the patterns in Table V. For the identified 423,503 links, we sample 384 links with 5% confidence interval to evaluate the accuracy of the identified links. The accuracy is 0.79. We show the distribution of different relationships in the identified links in Table IV. The distribution is similar with the sampled data in Fig. 2. For example, in the sampled data, fixed relationship is the largest one, which accounts for 41.37% of the issue units belonging to the six relationships. Similarly, in Table IV, 50.85% links are automatically identified as the fixed relationship. We calculate the correlation between the two distribution, the result is 0.95. Due to the similar distribution, we use the identified 423,503 links to analyze the influence of developers' linking behavior.

B. Revisit the linking behavior

Dependent Relationship. As shown in Table. IV, the fixing of 7.76% issue units (dependent relationship) heavily depend on other issue units within or across projects. In Fig. 4, we present the distribution of dependent relationship on distinct linking directions. Most of the dependent relationship happens in *pr*→*pr*, *i*→*i* and *i*→*pr*. In *pr*→*pr* (43.07%), developers complain that the problem of a pull request causes the failure of another pull request or the merge of a pull request is blocked by another pull request[7]. For

[7]https://github.com/Yelp/paasta/pull/1394

TABLE V: The relationships and their associated syntactic patterns.

Category	Syntactic Patterns	Link Direction
Depended relationship	(block*, 0), (cause*, -1), (depend*, -1), (upstream, 0), (downstream, 0), (introduced by, -1), (wait*, -1), (because of, -1), (side-effect, -1), (first fix*, 0), (first step, -1), (requir*, -1), (need, -1), (prefix, -1)	*i*→*i*, *i*→*pr*, *pr*→*i*, *pr*→*pr*
Duplicate relationship	(duplicate*, 0), (dupe, 0), (close*, -1), (in favor of, -1), (fixed, 0), (done, -1), (address*, 0),(supercede*, 0), (supercsede*, 0), (obsolete, -1), (replace, -1), (same, 0)	*i*→*i*, *pr*→*pr*
Relevant relationship	(relate*, -1), (similar, -1), (relevant, 0)	*i*→*i*, *i*→*pr*, *pr*→*i*, *pr*→*pr*
Enhanced relationship	(add*, -1), (modifi*, 0), (revert*, 0), (rebas*, 0), (revis*, 0), (improv*, -1), (continuation of, -1),(original issue, 0), (based on, -1), (supplement, -1)	*i*→*i*, *i*→*pr*, *pr*→*i*, *pr*→*pr*
Fixed relationship	(fix, 0), (done, -1), (address, 0), (submitted for, -1), (duplicate, 0), (close, 0), (resolv*, 0)	*i*→*pr*, *pr*→*i*
Reference relationship	(comment, 0), (discussion, 0), (motivat*, -1), (example, 0), (e.g, -1), (suggestion, 0), (background, 0) , (referenc*, -1)	*i*→*i*, *i*→*pr*, *pr*→*i*, *pr*→*pr*

example, in the issue unit `astropy/astropy#2363`, *Embray* commented "*(The issue) is probably a regression introduced by spacetelescope/PyFITS#23, incidentally, before this was fixed the comment entry wasn't preserved at all upon write*". In *i*→*i* (27.84%), the dependent relationship shows that an issue report is blocked/caused by a target issue report. In *i*→*pr* (16.22%), most links indicate the relationship that the bugs in the source issue reports are introduced by some target pull requests.

As complained by developers, the dependent relationship tends to delay the fixing of source issue units. The problem is severer when two issue units are in different projects [6]. Hence, a mechanism to identify the dependent relationship may facilitate developers fixing both source and target issue units.

Finding 4. The resolution of 7.76% issue units heavily depend on other issue units within or across projects. A mechanism to identify the dependent relationship may facilitate developers' work.

Duplicate Relationship. There are 70,686 links with duplicate relationship in the identified links, which account for 16.69%. Fig. 3 shows the time span between submitting an issue unit and the time of first labeling it as "duplicate". For 15.16% of these issue units, submitters seem to already aware that the current issue units are duplicate as soon as they are submitted. Hence, submitters directly link these issue units to the possibly duplicate ones. We find that 79.35% of these links usually contain some uncertain words in their context, e.g., "possible", "may", "unsure". It means that most submitters are still unsure whether the duplicate relationship is true. Besides, 59.46% linking behavior happened in a relatively short time ($<$ 1 day). However, 40.54% issue units still have a long latency period ($>$ 1 day) before duplicate detection. Notably, 27.96% issue units were first marked as "duplicate" after 10 days. For these issue units, developers are more likely wasting their time in solving already fixed issue units.

For the 70,686 links with duplicate relationship, 59.03% are duplicate issues (*i*→*i*) and 40.97% are duplicate pull requests (*pr*→*pr*) (in Fig. 4). Considering that existing studies for duplicate detection mainly focused on detecting duplicate issue reports [7], the statistic shows that detecting duplicate pull requests is also meaningful, as it saves developers time in reviewing and testing these pull requests.

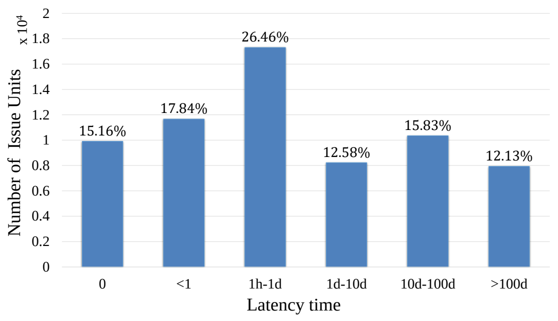


Fig. 3: Distribution of latency time (h:hour, d:day)

Finding 5. 40.54% of issue units are first marked as "duplicate" in more than one day. Notably, 27.96% issue units are first marked after 10 days. Detecting duplicate issue reports and pull requests are both important.

In addition, previous studies usually detect duplications within a single project [17], e.g., Eclipse, Mozilla. However, as a software ecosystem, the duplicate relationship may happen across projects, i.e., developers in different projects submit duplicate issues that caused by the same upstream projects. 5.97% links with duplicate relationship are related to different projects. Since these issue units refer to the same root cause, the source issue units may provide more evidence for upstream projects to fix root issue reports or test upstream pull requests. Hence, identifying duplication in an ecosystem is promising for study.

Finding 6. 5.97% duplicate relationship is related to different projects. It is promising to conduct duplicate detection from the point of a software ecosystem view.

Relevant Relationship. Relevant Relationship mainly appears in *i*→*i*(60.9%). In relevant relationship, the source and target issue units may discuss similar bugs, share the same topic, or similar solution. By manually analyzing relevant

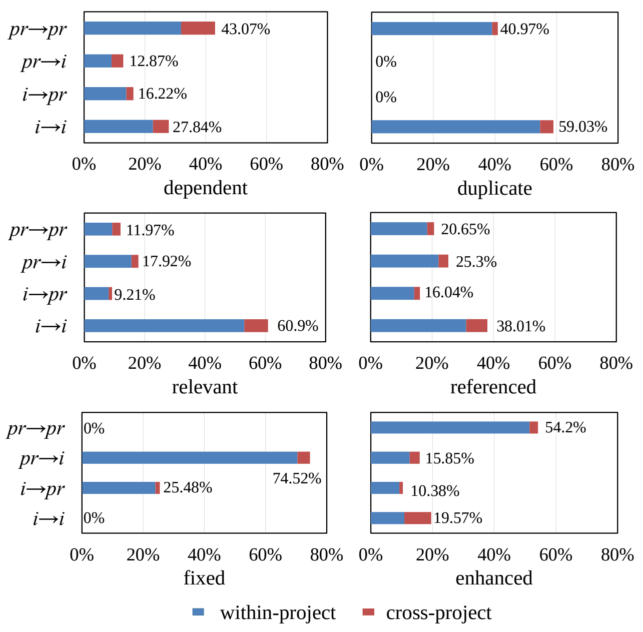


Fig. 4: Distribution of automatically identified relationships.

issue units, we find that developers prefer to studying relevant issue units when resolving new issue units. The links facilitate the process of bug fixing and make developers more efficient in handling a series of issue units [18]. For examples, in `beetbox/beets#1228`, *sampsyo* pointed out "*Hey, @brunal this looks similar to #1226.*". Then developers discuss `beetbox/beets#1226` and quickly solve the problem.

Referenced Relationship. In Table. IV, the referenced relationship accounts for 11.53%. It seems that developers are likely to quoting an existing issue unit, in order to provide additional information for the current problem or promote the current communication. The information provided by referenced relationship is different from the relevant relationship. Relevant issues let developers understand specific solution and facilitate the process of bug fixing, while referenced issue is used to help developers understand specific concepts, background and motivations etc. As shown in Fig. 4, the referenced relationship mainly happens in *i→i* (38.01%) and *pr→i* (25.3%). In *i→i*, the target issue units may play the role as background, motivation, example, concept explanation of the source issue units. In *pr→i*, developers link a target issue unit to explain the source code and methods for the source issue unit. According to the results of automatic classification, 63.04% referenced relationship is identified by the patterns of 'comment*' and 'discuss*'. By manually investigating the large-scale matched sentences, we find that developers usually use the sentences like "*as discussed in #Num*", "*see the comment in #Num*", etc. Hence, it reconfirms that historical issue units contain tremendous knowledge for developers to address new issue units.

Finding 7. 63.04% links in referenced relationship are recommending developers to refer to historical comments or discussion. It reconfirms that historical issue units contain tremendous knowledge to address the new ones.

Fixed Relationship. We detect fixed relationship in *pr→i* and *i→pr*. In *pr→i*, the fixed relationship means the developer submits a pull request to fix an issue report. In *i→pr*, developers link an issue report to a pull request because the pull request has totally or partially addressed the problem in the issue report. As shown in Table. IV, the fixed relationship is a predominant relationship, which accounts for more than 50.85% links in all the identified links.

For the fixed relationship, we find that the in-degree of an IUN is an indicator to reveal the importance (priority) of the fixed issue units. The fixed issue units with high priority are inclined to have greater in-degree in the IUN. To prove this finding, we split the issue units in our data set into high-priority and low-priority ones by analyzing the labels of the issue units. High-priority issue units are usually associated with the labels of "high (priority/server)", "critical", and "urgent". In contrast, low-priority issue units are labeled by "low (priority/server)". In this way, we discovered 6,318 high-priority issues units and 6,120 low-priority ones. We calculate the in-degree of each collected issue units in the IUN. The mean values of the in-degree for the high-priority and low-priority issue units are 0.44 and 0.24 respectively. Clearly, high priority issue units have greater in-degree than the low priority ones. The observation passes the Wilcoxon-Mann-Whitney test [19] with p=1.94e^{-24}, i.e., the in-degree of a randomly selected issue unit in the high-priority sample is usually greater than that in the low-priority sample.

Hence, an IUN can help new comers of a project identify the core issue units in the project. The new comers may better understand a project by studying these issue units.

Finding 8. The in-degree of issue units is an factor to indicate their importance. By analyzing IUN, new comers can identify and study the core issue units of a project.

Enhanced Relationship. About 5.27% linking behavior belongs to the enhanced relationship. Among these links, 54.20% enhanced relationship happens in *pr→pr*. In *pr→pr*, an enhanced relationship usually means to revert, revise, or rebase a pull request. This may cost developers additional effort on a pull request. Hence, we manually analyze the reasons to enhance a pull request in the sampled data in Section IV. The main reasons are the incompleteness of the pull requests, including the missing of new features, test scripts, logging statements, and documentation. These reasons account for 59.32% in our sample for *pr→pr*. Hence, developers will enhance the issue unit to make it meet the requirements. We suggest that developers can pay close attention to the issue units of enhance relationship, in order to learn how to submit a good pull request.

Finding 9. 54.20% enhanced relationship happens in *pr→pr*. There are many reasons to enhance the pull requests, e.g., the incompleteness of test scripts, logging statements, etc.

VI. Threats to Validity

Our work is subject to several validity problems, including the threats to IUN construction, manual classification, and automatic classification.

IUN construction. To construct IUN, we identify links with patterns "*#Num*" and "*User/Project#Num*". However, not all the links identified by these patterns are valid. First, the pattern "*#Num*" may just indicate an natural number instead of an issue unit ID. To mitigate these false positive links, we use GitHub APIs to verify whether each number can reach a valid issue unit. Second, as suggested by Kalliamvakou et al. [20], one peril in mining GitHub repository is "a repository is not necessarily a project", i.e., a repository may be a fork of a base repository, which will affect our judgments on cross-project links. To avoid this peril, we define a base repository with its forked repositories as the same project. In other word, we consider a link from a forked repository to a base repository as a within-project link. Hence, we recognize cross-project links by comparing the name of source projects with the name of both target projects and the parent of target projects. A sampling validation shows that we achieve an accuracy value of 92% in identifying issue unit links.

Manual Classification. Manual classification involves two threats. The first one is the representativeness of the sampled data, i.e., the 1,384 randomly sampled links. We addressed this threat by performing the Stratified sample considering a confidence level of 95% over different link directions to ensure the data representativeness. Meanwhile, we only focus on the main reasons for linking behavior. According to the results of automatic classification, the main reasons in the sampled data represent at least 44% linking behavior in the IUN. The second threat is the mistakes in manual classification. Since manual classification is subjective, it is hard to develop a perfect coding schema with no conflicts among the categories. To reduce the mistakes, we perform a three-stage classification by two volunteers. The consistent rate is 0.883. Hence, the rules for classification achieve reliable results.

Automatic Classification. In this study, we automatically classify the linking behavior of high-quality Python projects. Several findings depend on the classification results. However, the automatic classification may contain errors. Hence, we validate a sample of classification results, which shows an accuracy of 79%. Besides, we mainly draw conclusions from the major classes, since the error instances may dominate when the instance number of a class is small. We note that our automatic classification only cover about half of linking behavior, i.e., 423,503 links. Hence, our findings may be only valid on these links. Since the distribution of the identified links is similar to the Stratified sampled data, we may draw consistent findings when sampling another set of issue units for analysis. In the future, we plan to improve the performance of automatic classification and re-verify our findings on more data. We also plan to automatically analyze the relationship between a newly submitted issue unit and all the historical ones to help developers immediately find dependent, duplicate, referenced issue units.

VII. Related Work

Our work is related to two research lines, including the studies on issue units and the analysis on linked network.

A. Studies on Issue Units

Many studies explore the knowledge of issue units in GitHub. Issue units include the issue reports and pull requests. For issue reports in Github, researchers predict the attributes of issue reports, such as the labels of issue reports [21], the "mentions" be used in issue reports [22]. Some studies also analyze the correlation between the process of reporting issue reports and the success of software projects [23].

For pull requests, most previous studies focus on the reviewer recommendation for pull requests. They recommend code reviewers by analyzing the working history of a developer and collaboration history with other developers, including the line change history [24], review comments [25], [26], project directory structure [27] [28], developer collaboration network [26], developer communication history, and cross-project experience [29]). Besides the above studies, many empirical studies are also conducted with GitHub data. Gousios et al. and Saito et al. explore how pull request based software development works and how GitHub users feel with this software development model respectively [4] [30]. The factors that affect the acceptance of pull request have also been studied in previous studies [31]–[33].

B. Linked Network Analysis

Recently, many studies investigate the knowledge network for software development. In the social platforms, e.g., Stack Overflow, Ye et al. and Xu et al. construct a large knowledge network with the internal URLs in Stack Overflow posts [16] and predict semantically linkable knowledge units using neural language model and convolutional neural network (CNN) [34]. In issue tracking system, Correa et al. investigate the online resources in Google Chromium Browser project [35]. They studied the importance, distribution, and categorization of links in issue reports. Similarly, links between pull requests were researched by Zampetti et al [36], which investigates to what extent and for which purpose developers refer to external online resources.

In this study, we focus on the issue tracking system and construct IUN with GitHub data. However, previous studies only investigate the links to external online resources, i.e., the links to other web sites. We find that developers also frequently refer to the knowledge within the web site, e.g., the historical issue units. We analyze the reasons for these linking behavior. By analyzing the issue units network, we provide a series of findings to promote software development and maintenance activities.

VIII. Conclusion

In this paper, we conducted an empirical study to explore developers' linking behavior in GitHub by analyzing the links they left within issue reports and PRs. Six major kinds of relationships (which well explained developers' linking behavior) were identified through manual analysis. We further proposed several common patterns that can help automatically classify linked issues/PRs into these relationship categories. At last, we analyzed the influence of different relationships on software development, and found several interesting findings, including the importance of detecting cross-project duplicate bug reports, the potential of using IUN to identify influential projects and core issue reports, etc. These findings, to some extent, provided a basis for researchers and relevant practitioners to develop practical techniques that help improve developers' experience in handling issues/PRs.

References

[1] G. Inc. (2018) About issues. [Online]. Available: https://help.github.com/articles/about-issues/

[2] ——. (2018) About pull requests. [Online]. Available: https://help.github.com/articles/about-pull-requests/

[3] C. C. M. Neto and D. O. B. Mrcio, "A structured survey on the usage of the issue tracking system provided by the github platform," in *Brazilian Symposium on Software Components, Architectures, and Reuse*, 2017, pp. 1–10.

[4] G. Gousios, M. Pinzger, and A. V. Deursen, "An exploratory study of the pull-based software development model," in *International Conference on Software Engineering*, 2014, pp. 345–355.

[5] G. Inc. (2017) The state of the octoverse 2017. [Online]. Available: https://octoverse.github.com/

[6] W. Ma, L. Chen, X. Zhang, Y. Zhou, and B. Xu, "How do developers fix cross-project correlated bugs? a case study on the github scientific python ecosystem," in *Ieee/acm International Conference on Software Engineering*, 2017, pp. 381–392.

[7] Z. Li, G. Yin, Y. Yu, T. Wang, and H. Wang, "Detecting duplicate pull-requests in github," in *Asia-Pacific Symposium on Internetware*, 2017, pp. 1–6.

[8] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: How misclassification impacts bug prediction," in *International Conference on Software Engineering*, 2013, pp. 392–401.

[9] R. Lotufo, Z. Malik, and K. Czarnecki, "Modelling the hurried bug report reading process to summarize bug reports," in *IEEE International Conference on Software Maintenance*, 2012, pp. 430–439.

[10] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th working conference on mining software repositories*. ACM, 2014, pp. 92–101.

[11] G. Inc. (2017) Github octoverse 2017. [Online]. Available: https://octoverse.github.com/

[12] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of github repositories," in *IEEE International Conference on Software Maintenance and Evolution*, 2017, pp. 334–344.

[13] K. Blincoe, F. Harrison, and D. Damian, "Ecosystems in github and a method for ecosystem identification using reference coupling," in *Mining Software Repositories*, 2015, pp. 202–207.

[14] J. Zhou and R. J. Walker, "Api deprecation: a retrospective analysis and detection method for code examples on the web," in *ACM Sigsoft International Symposium on Foundations of Software Engineering*, 2016, pp. 266–277.

[15] M. H. Hansen and W. N. Hurwitz, *Sample survey methods and theory. Vol. I.* John Wiley And Sons, Inc.; New York, 1953.

[16] D. Ye, Z. Xing, and N. Kapre, "The structure and dynamics of knowledge network in domain-specific q&a sites: a case study of stack overflow," *Empirical Software Engineering*, vol. 22, no. 1, pp. 1–32, 2017.

[17] G. Canfora, M. Cimitile, M. Cimitile, and M. D. Penta, "Social interactions around cross-system bug fixings: the case of freebsd and openbsd," in *Working Conference on Mining Software Repositories*, 2011, pp. 143–152.

[18] H. Rocha, M. T. Valente, H. Marques-Neto, and G. C. Murphy, "An empirical study on recommendations of similar bugs," in *IEEE International Conference on Software Analysis, Evolution, and Reengineering*, 2016, pp. 46–56.

[19] F. Dexter, "Wilcoxon-mann-whitney test used for data that are not normally distributed," 2013.

[20] Kalliamvakou, Eirini, Gousios, Georgios, Blincoe, Kelly, Singer, Leif, and M. Daniel, "The promises and perils of mining github," *Empirical Software Engineering*, vol. 37, no. 5, pp. 1–10, 2006.

[21] J. Cabot, J. L. Canovas Izquierdo, V. Cosentino, and B. Rolandi, "Exploring the use of labels to categorize issues in open-source software projects," in *IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2015, pp. 550–554.

[22] Y. Zhang, G. Yin, Y. Yu, and H. Wang, "A exploratory study of @-mention in github's pull-requests," in *Software Engineering Conference*, 2015, pp. 343–350.

[23] T. F. Bissyande, D. Lo, L. Jiang, and L. Reveillere, "Got issues? who cares about it? a large scale investigation of issue trackers from github," in *IEEE International Symposium on Software Reliability Engineering*, 2013, pp. 188–197.

[24] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *International Conference on Software Engineering*, 2013, pp. 931–940.

[25] X. Xia, D. Lo, X. Wang, and X. Yang, "Who should review this change?: Putting text and file location analyses together for more accurate recommendations," in *IEEE International Conference on Software Maintenance and Evolution*, 2015, pp. 261–270.

[26] Y. Yu, H. Wang, G. Yin, and C. X. Ling, "Reviewer recommender of pull-requests in github," in *IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 609–612.

[27] P. Thongtanunam, R. G. Kula, A. E. C. Cruz, N. Yoshida, and H. Iida, "Improving code review effectiveness through reviewer recommendations," in *International Workshop on Cooperative and Human Aspects of Software Engineering*, 2014, pp. 119–122.

[28] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K. Matsumoto, "Who should review my code? a file location-based code-reviewer recommendation approach for modern code review," in *IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2015, pp. 141–150.

[29] M. M. Rahman, C. K. Roy, and J. A. Collins, "Correct: code reviewer recommendation in github based on cross-project and technology experience," in *Ieee/acm International Conference on Software Engineering Companion*, 2016, pp. 222–231.

[30] Y. Saito, K. Fujiwara, H. Igaki, N. Yoshida, and H. Iida, "How do github users feel with pull-based development?" in *International Workshop on Empirical Software Engineering in Practice*, 2016, pp. 7–11.

[31] Y. Yu, H. Wang, V. Filkov, P. Devanbu, and B. Vasilescu, "Wait for it: Determinants of pull request evaluation latency on github," in *Mining Software Repositories*, 2015, pp. 367–371.

[32] D. M. Soares, M. L. D. L. Jnior, L. Murta, and A. Plastino, "Rejection factors of pull requests filed by core team developers in software projects with high acceptance rates," in *IEEE International Conference on Machine Learning and Applications*, 2016, pp. 960–965.

[33] L. Murta and A. Plastino, "Acceptance factors of pull requests in open-source projects," in *ACM Symposium on Applied Computing*, 2015, pp. 1541–1546.

[34] B. Xu, D. Ye, Z. Xing, X. Xia, G. Chen, and S. Li, "Predicting semantically linkable knowledge in developer online forums via convolutional neural network," in *Ieee/acm International Conference on Automated Software Engineering*, 2016, pp. 51–62.

[35] D. Correa, S. Lal, A. Saini, and A. Sureka, "Samekana: A browser extension for including relevant web links in issue tracking system discussion forum," in *Software Engineering Conference*, 2013, pp. 25–33.

[36] F. Zampetti, L. Ponzanelli, G. Bavota, A. Mocci, M. Di Penta, and M. Lanza, "How developers document pull requests with external references," in *Program Comprehension (ICPC), 2017 IEEE/ACM 25th International Conference on*. IEEE, 2017, pp. 23–33.