sustainability

MDPI

Article

What Is Needed for the Sustainable Success of OSS Projects: Efficiency Analysis of Commit Production Process via Git

Jaeyoon Song [1] and Changhee Kim [2,*]

[1] College of Business Administration, Seoul National University, Seoul 08826, Korea; song@jaeyoon.io
[2] College of Business Administration, Incheon National University, Incheon 22012, Korea
* Correspondence: ckim@inu.ac.kr; Tel.: +82-032-835-8734

Received: 21 June 2018; Accepted: 21 August 2018; Published: 23 August 2018

Abstract: The purpose of this study is to investigate the relative efficiency of open source software projects, and to analyze what is needed for their sustainable success. The success of open source software is known to be attributable to a massive number of contributors engaging in the development process. However, an efficient open source software project is not guaranteed simply by active participation by many; a coordination mechanism is needed to seamlessly manage the multi-party collaboration. On this basis, this study aimed to examine the internal regulatory processes based on Git and GitHub, which serve as such a mechanism, and redefine the efficiency of open source software projects to fully reflect them. For this purpose, a two-stage data envelopment analysis was used to measure the project efficiency reflecting the internal processes. Moreover, this study considered the Kruskal–Wallis test and Tobit regression analysis to examine the effects of the participation by many on an open source software project based on the newly defined efficiency. Results show that a simple increase in contributors can be poisonous in terms of the efficiency of open source software projects.

Keywords: open source software projects; collaboration; Git; GitHub; data envelopment analysis; efficiency

1. Introduction

Open source software (OSS) does not simply refer to software distributed for free. It is a concept that allows not only the actual usage of software, but also free redistribution, which means anyone can access and modify the original source code to produce derivative products for their sustainability [1]. Linux, MySQL, and Chrome are representative examples of sustainable and successful OSS projects. Moreover, OSS is being used by countless companies and government organizations; the number of OSS projects as well as the volume of the source code is on exponential growth on a global basis [2].

From a traditional point of view, the sustainable success of such OSS projects is rather unconventional. The online communication and cooperation of random people by voluntary motivation without relying on economic interests cannot be explained with the conventional economic viewpoint. Raymond [3] asserted "given enough eyeballs, all bugs are shallow" as 'Linus's Law', citing "a large enough beta-tester and co-developer base" at the core of the success of OSS. In other words, the massive number of contributors engaging in the testing, bug-fixing, and function-adding leads to a product that surpasses commercial software that is developed on an exclusive basis.

However, this free and open environment is not without limitations. This is because a large number of participants comparatively reduce group efficiency [4]. The larger the group becomes, the higher the potential for inefficient communication, which makes seamless project management difficult. For example, in terms of high-level decision making such as adding critical functions,

the participation by many becomes toxic in nature. In reality, many OSS projects fail without reaching maturity owing to inefficiency. According to Schweik and English [5], only 17% of total OSS projects end in a success.

Then, what is needed for an OSS project to be efficient? An efficient OSS project is not guaranteed simply by the active participation of many. For efficient projects, a coordination mechanism is needed to seamlessly manage the multi-party cooperation. In other words, the multiple parties collaborating on an OSS project must be regulated to a certain extent by a smaller number of managers with appropriate authority [6]. Git, covered in this study, is one example of such a regulatory device for voluntary collaboration.

Therefore, a reflection of such internal regulatory processes is indispensable for analyzing the efficiency of an OSS project. Simply reviewing the initial input and final output cannot demonstrate the efficiency of the project. Hence, this study aims to move away from the existing perspective to a new one, reflecting a series of processes accompanying regulated voluntary cooperation, and to redefine the efficiency of OSS projects. For this purpose, this study has set a two-stage model of data envelopment analysis (DEA) to reflect internal processes. Moreover, based on the newly defined efficiency, this study examines the effects of the participation by many on an OSS project through the Kruskal–Wallis test and Tobit regression analysis.

2. Background and Literature Review

Over many years, several scholars have taken an interest in the efficiency of OSS projects and have presented their own interpretations. Among such scholars, Ghapanchi and Aurum [7], Wray and Mathieu [8], and Koch [9] utilized DEA to measure relative efficiency.

First, the research model of Ghapanchi and Aurum [7] is presented in Figure 1. This study utilized the partial least square (PLS) method to determine the positive influence of the four competencies of the theory of competency rallying (TCR) on the OSS project performance. Within the context of OSS, TCR covers a process of rallying the individual developers with capabilities to respond to new customer needs. The four capabilities of TCR—identification of market needs, marshalling of competencies, development of competencies, and managing cooperative work—are the independent variables of the research model; the OSS project performance, composed of developer interest and project efficiency, is the dependent variable. DEA was used in the analysis of project efficiency—the second component of project performance—with the number of developers and the project duration as inputs, and the number of released files as output. The subjects of the analysis were projects on the OSS development platform Sourceforge.net; in order to secure homogeneity, 607 projects in the software development category were selected.

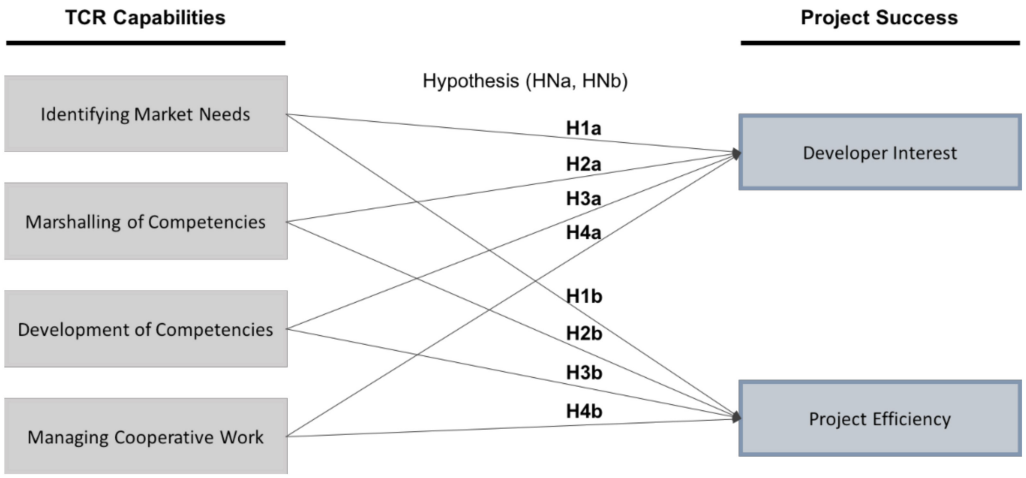

Figure 1. Research model of Ghapanchi and Aurum [7].

Wray and Mathieu [8] selected 34 security-based OSS projects in Sourceforge.net as decision making units (DMUs) and utilized an input-oriented Banker–Charnes–Cooper (BCC) model. The model considered the number of developers and number of bug submitters as input variables, and the project rank, number of downloads, and kilobytes per download as output variables. Particularly, this research provides several implications in that it included bug submitters as well as the number of developers in the inputs, and qualitative indicators in the outputs. The present study also aims to reference this aspect and approach inputs and outputs from a multidimensional perspective.

Koch [9] utilized DEA to understand the influence of cooperation tools on the efficiency of OSS projects. First, two types of DMU groups were selected; the first group was composed of the top 30 projects sorted by Sourceforge.net rankings, and the other group was randomly compiled. The inputs involved the number of developers and the number of years, and the outputs were downloads, web hits, size in bytes, and lines of code. Efficiency was measured using the output-oriented BCC model.

Such existing literature have contributed to newly defining the efficiency of OSS projects, which are clearly segregated from traditional closed-source software, but have limitations in two aspects.

The first limitation is limiting the research subject to Sourceforge.net projects, which presents a gap in the application of their results to the latest OSS projects. Sourceforge.net has now become a website for merely downloading software and no longer represents OSS platforms. This was largely due to GitHub, which started in 2007 and has now rapidly grown to be the world's largest OSS project platform. As GitHub came to dominate the OSS market, other OSS communities such as Google Code, fell behind and ended service in 2015. This study is based on GitHub in its measurements of OSS project efficiencies to derive results that are more appropriate in the present day.

The second limitation is that the internal process of the project was not considered in the analysis. GitHub offers a version control system called Git, which involves a more complex development process. OSS projects on GitHub are characterized by a participant writing code, creating a commit and undergoing a review process before being merged into the master branch through Git. The traditional DEA model is unable to accurately capture the efficiency of OSS projects with such complex processes. This is because the model regards the process between input and output as a black box [10]. Therefore, it is necessary to apply the expanded DEA model to take into account the internal processing mechanism of OSS projects. As such, this study utilizes a two-stage DEA method to identify internal inefficiencies, which are difficult to spot using the traditional DEA.

3. Materials

3.1. Git and GitHub

The subjects of this study were projects on GitHub, which is the worldwide platform for OSS development. GitHub provides a wide range of advanced functions that support cooperation for free, but the core rationale is Git—a version control system. A version control system is a system that manages all histories of source code changes so that multiple developers can simultaneously make changes to the same source code. In other words, it is a type of autonomous regulatory device to effectively manage the participation by many. Git saves each change of source code as a unit called a commit; the change is not applied immediately, but only after the code has passed a step-by-step examination through a series of mechanisms. OSS projects on GitHub are operated using Git by default.

GitHub explains the process of contributing to an OSS project using Git from the standpoint of a developer in six stages [11]:

1. The first task is to create a branch in the project that the developer wishes to contribute to. The place where the code of the OSS is to be actually released is called the master branch; branches are used to prevent confusion from all changes being made to the master branch. For instance, if one wants to propose a function relating to comments, one can work on a branch called 'comment' if such a branch exists. Moreover, branches are like actual branches stemming from the master and thus begin from the code of the master; however, unless the administrator

merges the branches, the changes to the code in a specific branch do not impact the master code. One can also create a fork of a project. To fork is to copy the said project. This is useful when testing the changes to the code, because the copied version of the project can be changed without influencing the original project.

2. The second stage is to create a commit by writing codes. This may relate to removing errors in the existing code or adding a new function.
3. The third stage involves writing a pull request, which shows the modified code segment with a simple explanation. In other words, a pull request is a message that requests the project administrator to accept—or pull—the changes to the original.
4. Next, the pull request is discussed and reviewed by contributors. Anyone can leave a comment with evaluation of the pull request.
5. The fifth stage involves the project administrators accepting the pull request or not. When the pull request is rejected, the code in question is not merged and thus is returned. If the pull request is accepted, the code is created as a commit of the branch.
6. Finally, when the branch is ready for distribution, it is merged into the master branch. Even after the merge, related issues or bugs may be reported. Issues are a communication channel on GitHub, and are a type of bulletin board where anyone can present opinions about the project.

Through the above process, Git and GitHub help to make the cooperation of an unspecified many more effective. This process can be largely subdivided into two parts, as shown in Figure 2. The first regulatory device limits the commits through pull requests, and is shown in Figure 2a; Figure 2b is a device that limits the changes to be reflected directly on the master through the branch. However, not all commits go through all of these processes in an OSS project. Project administrators have the authority to directly push commits, and they often bypass the evaluation process of pull requests. As such, all commit information in an OSS project can be divided into those pull requests that have passed an open evaluation, and commits that have been added by those with authority to directly add commits [6].

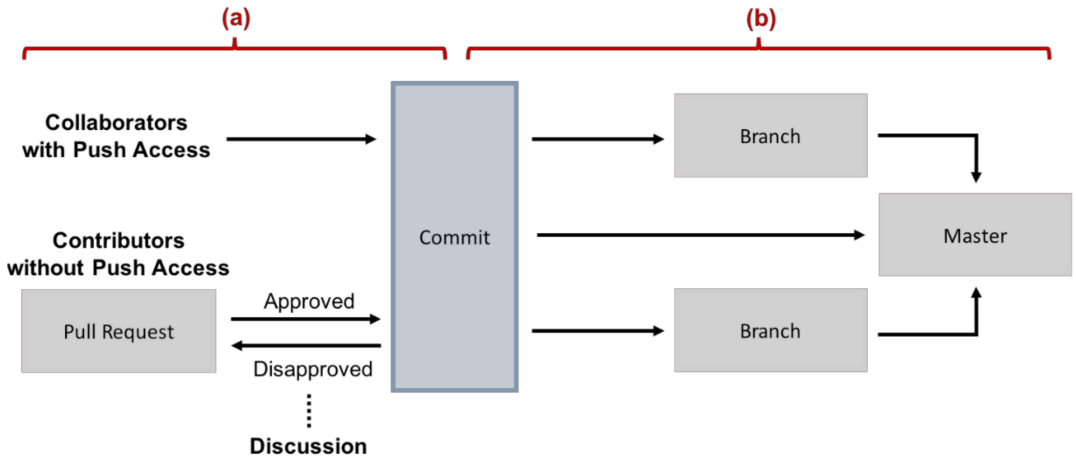

Figure 2. The process of commit production via Git: (**a**) coordination through the review of pull requests; (**b**) coordination by creating a separate branch stemming from the master.

3.2. Data Collection

In discussing the efficiency of the OSS project, this study aimed to reflect the process through which commits are generated and merged into the master branch. To achieve this, data were needed on a diverse range of features in the Git-based development process such as commit, branch, fork, pull request and issue; these can be collected using the GitHub application programming interface (API). GitHub provides a free API service, allowing external parties to access the accumulated data [12].

The collection of DMU information required for this research was done based on the classification of Showcase pages provided by GitHub; GitHub API crawling was used to collect the values corresponding to input, output and environmental variables.

4. Methods

4.1. DEA

Data envelope analysis (DEA) is a method that is used to measure the comparative efficiency of decision making units (DMUs) with multiple inputs and outputs [13]. The efficiency of software projects is usually measured with DEA. DEA is particularly useful for OSS whose production process is complex and difficult to define because DEA derives an efficient frontier using empirical data without explicitly assuming the production function and measures the efficiency of each object of evaluation based on its distance from the frontier [14]. Furthermore, DEA provides useful information on performance benchmarking, such as which effective DMU should be referenced to and how much improvement is needed in input or output elements.

To conduct efficiency analysis using DEA, an appropriate returns to scale assumption of the DEA model must be established. In the case of OSS projects, the assumption of variable returns to scale is applied because the scales are various and both increasing and decreasing returns to scale exist in the information technology industry [15]. As such, this study selected the Banker–Charnes–Cooper (BCC) model, which assumes variable returns to scale. The form of the equation used in this study is shown below:

$$\begin{aligned} \text{Maximize } h_0 = \quad & \sum_{r=1}^{s} u_r y_{r0} + u_0 \\ & s.t. \\ & \sum_{r=1}^{s} u_r y_{r0} - \sum_{i=1}^{m} v_i x_{ij} + u_0 \leq 0,\ j = 1, \ldots, n \\ & \sum_{i=1}^{m} v_i x_{i0} = 1 \\ & u_r, v_i \geq \varepsilon,\ \forall r,\ i \end{aligned} \tag{1}$$

4.2. Two-Stage DEA

OSS projects on GitHub are accompanied by a very complex development process. A traditional DEA assumes this commit production process as a single black box, and is unable to accurately reflect the efficiency of the GitHub OSS project. Therefore, this study utilizes the two-stage DEA model, which breaks down the internal structure of DMU in two stages rather than one, to analyze efficiency. The application of two-stage DEA can reveal internalized inefficiencies within a DMU which appears efficient in the traditional DEA model. Moreover, this model shows which stage specifically causes inefficiency since it calculates the efficiency of each stage independently; as such, guidelines to improve efficiencies can be specified by stage.

A commit carries a symbolic meaning in GitHub-based OSS development process. As mentioned earlier, a commit refers to an independent data unit that contains records of newly created code, which is created by Git [6]. It would be no exaggeration to refer to GitHub as a service that supports the convenient management and sharing of commits. This is because all code that makes up the OSS is contained within at least one commit. In discussing the efficiency of the OSS project, this study aimed to reflect the process through which the commits are created and merged into the master branch; as such, the mid-level output of the two-stage DEA model was set as the number of commits. On this basis, the merge efficiency of the first stage evaluated how efficiently the commits could be created by a large number of participants, and the project efficiency of the second stage evaluated the qualitative and quantitative outputs from the commits.

4.2.1. Stage 1—Merge Efficiency

All commits in an OSS project are divided into two types: those resulting from pull requests that have passed a phased regulatory phase, and those that have been directly added by contributors with push authority. As such, the merge efficiency of the first stage considers the number of pull requests and number of contributors as the input, and the number of commits as the output. As the OSS projects do not involve monetary costs for hiring human resources, but rather the voluntary participation of an unspecified many, it can be assumed that there is no control over inputs. Therefore, it is appropriate to use the output-oriented model, which seeks maximization of output for a given number of inputs [9].

Basically, to simplify the concept, merge efficiency gets lower by two reasons: lazy authority or an unacceptable pull request. The former is the case where collaborators with push authority do not produce enough commits, while the latter is the case where the majority of pull requests are unaccepted owing to problems such as low quality of the suggested codes.

4.2.2. Stage 2—Project Efficiency

In the second stage, project efficiency is analyzed with the number of commits as input and the number of stars and the project size as outputs. The project efficiency is characterized by relatively fixed input compared to the output and seeks for the maximization of outputs with the number of commits given; thus, the output-oriented model is selected.

4.3. DMU Selection

To properly apply the DEA model to a study of OSS efficiency, the compared DMUs must be homogeneous. Loss of homogeneity may cause the result to depend on external factors rather than on the objects of analysis [16]. For this reason, Ghapanchi and Aurum [7] narrowed the scope of samples to projects under the software development category among a variety of categories offered by Sourceforge.net. In addition, the majority of existing literature limited DMUs to a certain category such as ERP projects [17], Y2K projects [18], and security-based projects [8], and so on.

While there is no clear division of subject categories in GitHub unlike Sourceforge.net, it provides a collection of projects with popular subjects through its Showcase page [19]. Among 52 subjects provided by GitHub, this study obtains homogeneity by targeting the web application frameworks group as it has the largest number of projects that meet the following two conditions:

1. License

   The ownership and distribution rights for an individual OSS are specified under a diverse range of licenses. In this study, only the projects that contained License.md or License.txt files were selected to limit the objects of analysis to projects that officially correspond to OSS.
2. Activeness

   To secure the effectiveness of research, this study targeted only the projects that have been active until recently and fully utilize the features of Git. These were projects in which a commit was submitted within a month at the latest, at least 20 contributors were participating, and both pull requests and existing issues were selected.

GitHub presents the 29 projects under the subject of web application frameworks, which provides the framework for server-side web development. All 29 projects met the above two conditions and were selected as DMU. As this value is larger than three times the number of variables used, it is considered a valid number of DMUs [20].

4.4. Input and Output Variables

When measuring the efficiency of OSS projects, a common input is the number of developers, which is analogous with the labor cost in traditional projects [7–9]. Nevertheless, there are logical flaws in considering the number of contributors provided by GitHub as an input. The 'labor' of an

OSS project includes those writing code, as well as those reporting bugs through an issue or presenting opinions. However, the concept of a contributor defined by GitHub is limited to those whose code suggestion was actually accepted. In other words, bug reporters and those whose pull request was declined are ignored. This is because the level of contribution is determined by the number of commits on GitHub. Referring to Figure 2, all commits are created through two paths: pull requests and users with push access. Therefore, as inputs of OSS project efficiency, not only the number of contributors, but also the number of pull requests were selected. As shown in Table 1, correlation level between two variables were low enough, while there still existed a potential for multicollinearity.

Table 1. Correlation analysis between input variables.

	Number of Contributors	Number of Pull Requests
Number of Contributors	1	
Number of Pull Requests	0.6317	1

As the number of commits—the mid-level variable—only contains codes that were evaluated to be significant and accepted, it differs from the input variables that also include the codes that have not been accepted. Moreover, commits sometimes contain deletions of wrong or inefficient codes, as well as additions of new codes. Therefore, it is a concept that clearly differs from project size, which measures the quantitative outputs of a project.

In the case of outputs, project size was selected as a quantitative indicator of the output of the OSS project [9]. GitHub API measures the size on the basis of the entire repository including all of its history in kilobytes [12]. However, quantitative indicators do not fully reflect the characteristics of OSS and only show one aspect of outputs. As such, Wray and Mathieu [8] proposed the number of downloads and the project rank as a qualitative output. Although GitHub does not provide a specific ranking of OSS projects, it is possible to sort the projects based on the number of stars—a type of bookmarking feature—and number of watches—a feature to follow a certain project. 'Starred' projects are listed in the star section of the menu, whereas relevant notifications about the projects user is 'watching' are shown in the user dashboard. Both features are considered to be appropriate qualitative outputs as they involve evaluation of the quality of OSS by the end-users. Table 2 shows the results of correlation between outputs; the number of stars and the number of watches have a correlation of 0.8579, which is rather high. Therefore, along with the project size as a quantitative output, this study has selected the number of stars—a feature that is more easily accessible by users—as an output, and excluded the number of watches from the final outputs. Figure 3 shows the resulting research model.

Table 2. Correlation analysis between output variables.

	Project Size	Number of Stars	Number of Watches
Project Size	1		
Number of Stars	0.1486	1	
Number of Watches	0.1567	0.8579	1

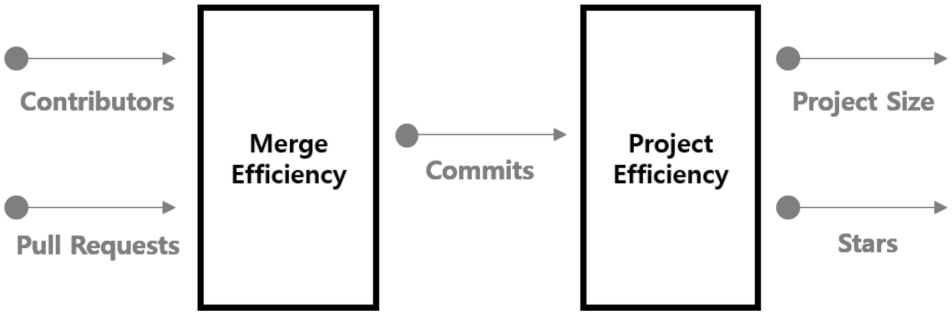

Figure 3. Research model based on two-stage DEA.

5. Results and Discussion

5.1. Stage 1—Merge Efficiency

Table 3 shows the results of a DEA performed with the output-oriented BCC model for the merge efficiency of 29 OSS projects in total. The inputs are the number of contributors, and the number of pull requests; the output of this stage—at the same time the mid-level variable of the entire model—is the number of commits (Y1). As this study utilized the output-oriented model, the output shortage values—rather than input surplus—are given in the corresponding columns of Table 5.

Table 3. Results of DEA analysis conducted on the merge efficiency.

NO.	DMU	Efficiency	Output Shortage	Reference Set	Reference Frequency
		VRS	Y1		
1	meteor	1	0	1	12
2	rails	1	0	2	5
3	laravel	0.2569	15,508.752	1,14	0
4	express	0.4373	6924.834	1,23	0
5	flask	0.2056	11,311.857	1,23	0
6	django	0.6466	13,344.874	2,14	0
7	sails	0.5276	5922.928	1,23	0
8	symfony	0.6064	20,318.942	2,14	0
9	spring–framework	0.9569	664.784	1,14,23	0
10	mean	0.1541	10,050.852	1,23	0
11	playframework	0.2986	19,386.406	1,14	0
12	sinatra	0.3171	8184.569	1,23	0
13	revel	0.1302	9046.574	23,29	0
14	cakephp	1	0	14	8
15	generator–jhipster	0.5242	9862.113	1,14,23	0
16	Nancy	0.3697	9086.848	1,23	0
17	nodal	0.328	1575.771	2,19,23	0
18	derby	0.6756	827.057	19,23,29	0
19	whitestorm.js	1	0	19	6
20	hanami	0.172	6795.926	23,29	0
21	padrino–framework	0.4326	7165.884	1,23	0
22	aspnetboilerplate	0.5693	2448.926	19,23,29	0
23	mojo [1]	1	0	23	18
24	ninja	0.2859	4825.71	2,19,23	0
25	kemal	0.1555	3031.334	19,23,29	0
26	web2py	0.5454	5977.949	1,14,23	0
27	pakyow	0.7804	383	19	0
28	frappe	0.7625	3926.326	2,14,23	0
29	catalyst–runtime	1	0	29	5

[1] Most frequently referenced projects in terms of merge efficiency.

Based on variable returns to scale (VRS) efficiency, six projects out of 29 were efficient and the efficiency was 1. The projects with relatively lower efficiency had an efficiency value lower than 1. The data on reference sets refer to the proposed benchmark targets, and reference frequency refers to the number of times the project was referenced as a benchmark target by other DMUs. With respect to merge efficiency, 'mojo' and 'meteor' were most frequently used as references, 18 and 12 times respectively, followed by 'cakephp', 'rails', and 'catalyst–runtime'.

5.2. Stage 2—Project Efficiency

In the case of project efficiency, as shown in Table 4, seven projects were found to be efficient out of 29, based on VRS. A noticeable finding is that aside from 'meteor', 'rails', and 'whitestorm' which were efficient in both stages, all four of project with the value of 1 for project efficiency—'nodal', 'laravel', 'flask', and 'kemel'—performed extremely poorly in merge efficiency. This indicates that the merge efficiency of the four projects needs to be improved to maximize the overall efficiency. Similar results can be drawn for 'cakephp', 'mojo', and 'catalyst–runtime', which had high merge efficiency but low project efficiency. This would not have been discovered if the analysis were done using the traditional DEA model, rather than measuring efficiency with the two-stage model, taking into account the internal processes.

Table 4. Results of DEA analysis conducted on the project efficiency.

No.	DMU	Efficiency	Output Shortage		Reference Set	Reference Frequency
		VRS	Y1	Y2		
1	meteor	1	0	0	1	13
2	rails	1	0	0	2	7
3	laravel	1	0	0	3	8
4	express	0.992	59.543	259.033	1,3,19	0
5	flask	1	0	0	5	8
6	django	0.9926	1220.522	195.024	1,2,19	0
7	sails	0.5343	10,056.401	14,958.264	1,3,19	0
8	symfony	0.5552	77,731.091	11,658.52	1,2,19	0
9	spring–framework	0.5975	68,153.278	9564.7	1,2,19	0
10	mean	0.6543	3520.564	5317.252	5,17,19	0
11	playframework	0.4779	94,823.088	10,228.94	1,3,19	0
12	sinatra	0.3307	12,287.866	18,892.354	3,5,19	0
13	revel	0.7871	637.302	2243.032	5,17,19	0
14	cakephp	0.3313	143,009.194	14,340.157	1,2,19	0
15	generator–jhipster	0.2376	66,319.186	22,696.612	1,3,19	0
16	Nancy	0.2773	117,077.073	13,752.152	1,3,19	0
17	nodal	1	0	0	17	6
18	derby	0.2985	5939.06	10,013.476	5,17,19	0
19	whitestorm.js [1]	1	0	0	19	22
20	hanami	0.4306	33,148.564	5232.87	5,17,19	0
21	padrino–framework	0.1198	79,127.997	22,304.609	1,3,19	0
22	aspnetboilerplate	0.1551	129,439.78	14,926.94	3,5,19	0
23	mojo	0.1088	194,344.026	14,961.117	1,2,19	0
24	ninja	0.183	174,667.035	7529.749	5,17,19	0
25	kemal	1	0	0	25	0
26	web2py	0.1438	229,045.323	8342.119	1,2,19	0
27	pakyow	0.0782	27,848.634	8887.761	5,17,19	0
28	frappe	0.3301	204,049	3351	19	0
29	catalyst–runtime	0.0449	282,982.276	5301.145	1,2,19	0

[1] Most frequently referenced projects in terms of project efficiency.

5.3. Overall Efficiency

There are two methods for deriving the overall efficiency to total the results of each stage in the two-stage DEA model: the additive method and the multiplicative method [21]. In the case of additive approach, the total value is calculated using a weighted average. However, there are numerous ways for calculating the weights, and it is difficult to appropriately apply such calculations based on the characteristics of OSS projects; thus, this study concluded that using weighted averages, which are only estimated values, could reduce the accuracy of the analysis. Therefore, overall efficiency was derived using the multiplicative method introduced in Kao and Hwang [22].

Furthermore, black box efficiency was calculated using the initial input and final output, considering the middle stage as a black box under the traditional DEA method. Comparing these figures with those of overall efficiency, the efficiency resulting from the two-stage DEA model was either equal to or lower than the black box efficiency. This is because the two-stage DEA can specify the development process of OSS projects to capture internal inefficiencies. The idea of efficiency decomposition is proven to have advantages over the traditional one-stage model in terms of systems that are accompanied by complex sub-processes [22,23]. One example that reconfirms the limitation of traditional DEA model is 'express'. Under the traditional model, the efficiency value of 'express' is 1; however, the overall efficiency or efficiencies of each stage of the same project reflect internal inefficiencies as shown in Table 5.

Table 5. Results of overall efficiency analysis.

No.	DMU	Two-Stage Efficiency			Black Box Efficiency
		Merge Efficiency	Project Efficiency	Overall Efficiency	
1	meteor	1	1	1	1
2	rails	1	1	1	1
3	laravel	0.2569	1	0.2569	0.8603
4	express [1]	0.4373	0.992	0.433802	1
5	flask	0.2056	1	0.2056	0.8106
6	django	0.6466	0.9926	0.641815	0.9626
7	sails	0.5276	0.5343	0.281897	0.5461
8	symfony	0.6064	0.5552	0.336673	0.522
9	spring–framework	0.9569	0.5975	0.571748	0.6036
10	mean	0.1541	0.6543	0.100828	0.371
11	playframework	0.2986	0.4779	0.142701	0.4404
12	sinatra	0.3171	0.3307	0.104865	0.3166
13	revel	0.1302	0.7871	0.10248	0.4631
14	cakephp	1	0.3313	0.3313	0.3424
15	generator–jhipster	0.5242	0.2376	0.12455	0.21
16	Nancy	0.3697	0.2773	0.102518	0.2467
17	nodal	0.328	1	0.328	0.9251
18	derby	0.6756	0.2985	0.201667	0.8527
19	whitestorm.js	1	1	1	1
20	hanami	0.172	0.4306	0.074063	0.2815
21	padrino–framework	0.4326	0.1198	0.051825	0.1162
22	aspnetboilerplate	0.5693	0.1551	0.088298	0.2764
23	mojo	1	0.1088	0.1088	0.1227
24	ninja	0.2859	0.183	0.05232	0.1654
25	kemal	0.1555	1	0.1555	0.2342
26	web2py	0.5454	0.1438	0.078429	0.1443
27	pakyow	0.7804	0.0782	0.061027	0.1905
28	frappe	0.7625	0.3301	0.251701	0.3301
29	catalyst–runtime	1	0.0449	0.0449	0.0558

[1] An example showing the limitation of traditional DEA model.

5.4. Determinants of Efficiency

"Given enough eyeballs, all bugs are shallow."

This is a quote from Raymond [3] that is known as 'Linus' Law'. This interprets the core principle behind a successful OSS as "many eyeballs". OSS projects—the voluntary collaboration of a massive number of anonymous developers around the world—often seems idealistic. In fact, OSS development surfaced as a rebellion against the monopoly of commercial software, valuing freedom and equality, and taking place through contribution and sharing by the crowd.

However, the participation by many can often be toxic. This is because the larger the group becomes, the potential for inefficient communication increases, which makes seamless project management difficult. The increase of elite participants and the production monopoly by the few is required for efficient OSS projects, rather than a simple increase in average participants [6].

Thus, in order to examine the relationship of the "participation by many" with the value of merge efficiency derived in Stage 1 of the two-stage DEA, this study considered the Kruskal–Wallis test and Tobit regression analysis. First, the Kruskal–Wallis test was conducted on two simple hypotheses. The Kruskal–Wallis test is an expanded model of the Mann–Whitney U test, and allows analysis of three or more groups. The test results are presented below.

According to the first hypothesis—H1 in Table 6—it is more efficient if the ratio of the number of contributors to the total number of commits is low. Higher number of contributors means more pull requests were accepted as commits because additional commits made by a collaborator with direct push access do not increase the number of contributors unless it is his or her very first commit. This indicates that the commits directly made by ones with push authority, instead of going through pull requests, act positively in terms of efficiency. In other words, closed production by a small number of authorities can raise the efficiency of an OSS project, more than the open participation of many. Through this observation, it can be concluded that measures to lower the ratio of the number of contributors to the number of commits should be considered for DMUs such as 'nodal', 'laravel', 'flask', and 'kemal', which have high project efficiency but extremely low merge efficiency. This can be done by active commit production of the small number of authorities, rather than anticipating a mere increase in total contributors—the increase in pull requests—to enhance the efficiency.

Table 6. Results of Kruskal–Wallis test.

Hypothesis	Sig.	Decision
H1. The lower the ratio of the number of contributors to the total number of commits, the higher the merge efficiency.	0.017	Accepted (Null hypothesis rejected)
H2. The higher the number of issues, the higher the merge efficiency.	0.108	Rejected (Null hypothesis retained)

According to the second hypothesis—H2 in Table 6—a higher number of issues does not relate to higher efficiency in OSS projects. This indicates that the participation by an excessive number of people may bring difficulty in communication, which would negatively affect the project. In order to ensure there is no influence of the project size as a potential confounding variable, a correlation analysis was performed between the project size and the number of issues. The resulting correlation value was 0.3558, which is low enough.

Furthermore, the Tobit regression analysis was conducted to estimate potential effects of different factors on OSS efficiency. The examined factors include the number of forks, heavy contributors, watches, issues, and branches. A heavy contributor was defined as a contributor who had created more than 100 commits. The regression results are shown in Table 7.

Table 7. Results of Tobit regression analysis.

	Dependent Variable: Merge Efficiency		Dependent Variable: Black Box Efficiency	
	Coefficient	**Standard Error**	**Coefficient**	**Standard Error**
(Intercept)	0.462	0.088	0.311	0.075
Forks	0.000	0.000	0.000	0.000
Heavy contributors	0.028 [1]	0.012	−0.009	0.010
Watches	0.000	0.000	0.000	0.000
Issues	0.000	0.000	0.000	0.000
Branches	0.001	0.001	0.001	0.001
Log-Sigma	−1.174	0.155	−1.317	0.146

[1] $p < 0.05$.

It can be seen from Table 7 that a higher number of heavy contributors leads to higher merge efficiency. These heavy contributors mostly correspond to a small number of collaborators with direct push access. Even if some of these collaborators are without push authority, the heavy contributors are clearly distinct from normal contributors as the proportion of heavy contributors is far too small; the heavy contributors accounted for only 4% of the total number of contributors on average. Thus, the results indicate that the increase in a few elite participants positively affects the efficiency, while a simple increase in the number of total contributors negatively affects the efficiency, as shown in Table 6. An additional regression analysis was conducted with the traditional black box efficiency set as the dependent variable. In contrast with the merge efficiency, black box efficiency did not show a statistically significant correlation with any of the examined factors, including the number of heavy contributors. This difference can be said to support the significance of utilizing the two-stage DEA model in the efficiency analysis of OSS projects.

6. Conclusions

This study reviewed the efficiency of open source software projects from the perspective that they are clearly different than commercial software, and introduced a mid-level variable to reflect the concepts of Git and GitHub, which work as a coordination system at the core of OSS. Furthermore, based on the derived results of efficiency and hypothesis testing, this study concluded that a small number of elite participants positively affects the efficiency, contrary to a simple increase in contributors that often has negative influences on the efficiency of OSS projects.

However, this study had a few limitations in data collection. For example, the output 'project size' is a value that developers often seek to minimize. This is in contradiction to the traditional point of view, which seeks to maximize the quantitative value of the output. To resolve this contradicting relationship, Wray and Mathieu [8] presented a new concept of "useful" project size, which is project size divided by the number of downloads. However, this is a variable that focuses on Sourceforge.net, where download was an important function. In GitHub, the number of downloads carries little meaning. This is because users often use package managers such as node package manager (NPM) or Bower when utilizing OSS.

Author Contributions: J.S. conceived and designed the experiments. The experiment was performed by J.S. and supervised by C.K. The paper was written by J.S. and revised by C.K. Finally, all authors read and approved the final manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Kechagia, M.; Spinellis, D.; Androutsellis-Theotokis, S. Open Source Licensing across Package Dependencies. In Proceedings of the 14th Panhelenic Conference on Informatics, Los Alamitos, CA, USA, 10–12 September 2010; pp. 27–32.

2. Da Silva, A.C.B.G.; Carneiro, G.D.F.; Abreu, F.B.E.; Monteiro, M.P. Frequent Releases in Open Source Software: A Systematic Review. *Information* **2017**, *8*, 109. [CrossRef]
3. Raymond, E.S. *Cathedral and the Bazaar*; SnowBall Publishing: La Vergne, TN, USA, 2010.
4. Steiner, I.D. *Group Process and Productivity*; Academic Pr.: New York, NY, USA, 1973.
5. Schweik, C.M.; English, R.C. *Internet Success*; The MIT Press: Cambridge, MA, USA, 2012.
6. Won, I. Collaboration and Control in Github. Master's Thesis, Seoul National University, Seoul, Korea, 2014.
7. Ghapanchi, A.H.; Aurum, A. The impact of project capabilities on project performance: Case of open source software projects. *Int. J. Proj. Manag.* **2012**, *30*, 407–417. [CrossRef]
8. Wray, B.; Mathieu, R. Evaluating the performance of open source software projects using data envelopment analysis. *Inf. Manag. Comput. Secur.* **2008**, *16*, 449–462. [CrossRef]
9. Koch, S. Exploring the effects of SourceForge.net coordination and communication tools on the efficiency of open source projects using data envelopment analysis. *Empir. Softw. Eng.* **2008**, *14*, 397–417. [CrossRef]
10. Tone, K.; Tsutsui, M. Dynamic DEA with network structure: A slacks-based measure approach. *Omega* **2014**, *42*, 124–131. [CrossRef]
11. Understanding the GitHub Flow GitHub Guides. Available online: https://guides.github.com/introduction/flow/ (accessed on 16 May 2018).
12. GitHub API v3 | GitHub Developer Guide. Available online: https://developer.github.com/v3/ (accessed on 16 May 2018).
13. Charnes, A.; Cooper, W.W. Programming with linear fractional functionals. *Nav. Res. Log. Q.* **1962**, *9*, 181–186. [CrossRef]
14. Banker, R.D.; Charnes, A.; Cooper, W.W. Some Models for Estimating Technical and Scale Inefficiencies in Data Envelopment Analysis. *Manag. Sci.* **1984**, *30*, 1078–1092. [CrossRef]
15. Kitchenham, B.; Pfleeger, S.; Pickard, L.; Jones, P.; Hoaglin, D.; Emam, K.E.; Rosenberg, J. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Eng.* **2002**, *28*, 721–734. [CrossRef]
16. Sarkis, J. Preparing Your Data for DEA. In *Modeling Data Irregularities and Structural Complexities in Data Envelopment Analysis*; Springer: Berlin, Germany; pp. 305–320.
17. Stensrud, E.; Myrtveit, I. Identifying high performance ERP projects. *IEEE Trans. Softw. Eng.* **2003**, *29*, 398–416. [CrossRef]
18. Yang, Z.; Paradi, J. DEA Evaluation of a Y2K Software Retrofit Program. *IEEE Trans. Eng. Manag.* **2004**, *51*, 279–287. [CrossRef]
19. Explore GitHub. Available online: https://github.com/explore (accessed on 21 September 2017).
20. Cooper, W.W.; Seiford, L.M.; Tone, K. *Data Envelopment Analysis: A Comprehensive Text with Models, Applications, References and DEA-Solver Software*; Kluwer Academic Publishers: Boston, MA, USA, 2002.
21. Yang, C.-H.; Lin, H.-Y.; Chen, C.-P. Measuring the efficiency of NBA teams: Additive efficiency decomposition in two-stage DEA. *Ann. Oper. Res.* **2014**, *217*, 565–589. [CrossRef]
22. Kao, C.; Hwang, S.-N. Efficiency decomposition in two-stage data envelopment analysis: An application to non-life insurance companies in Taiwan. *Eur. J. Oper. Res.* **2008**, *185*, 418–429. [CrossRef]
23. Sexton, T.; Lewis, H. Two-Stage DEA: An Application to Major League Baseball. *J. Prod. Anal.* **2003**, *19*, 227–249. [CrossRef]