GitHub Projects. Quality Analysis of Open-Source Software

Oskar Jarczyk[1], Błażej Gruszka[1], Szymon Jaroszewicz[2,3], Leszek Bukowski[1], and Adam Wierzbicki[1]

[1] Polish-Japanese Institute of Information Technology
Department of Social Informatics
Warsaw, Poland
{oskar.jarczyk,blazej.gruszka,bqpro,adamw}@pjwstk.edu.pl
[2] National Institute of Telecommunications
Warsaw, Poland
s.jaroszewicz@itl.waw.pl
[3] Institute of Computer Science
Polish Academy of Sciences
Warsaw, Poland

Abstract. Nowadays Open-Source Software is developed mostly by decentralized teams of developers cooperating on-line. GitHub portal is an online social network that supports development of software by virtual teams of programmers. Since there is no central mechanism that governs the process of team formation, it is interesting to investigate if there are any significant correlations between project quality and the characteristics of the team members. However, for such analysis to be possible, we need good metrics of a project quality. This paper develops two such metrics, first one reflecting project's popularity, and the second one - the quality of support offered by team members to users. The first metric is based on the number of 'stars' a project is given by other GitHub members, the second is obtained using survival analysis techniques applied to issues reported on the project by its users. After developing the metrics we have gathered characteristics of several GitHub projects and analyzed their influence on the project quality using statistical regression techniques.

Keywords: OSS, online collaboration, performance metrics, survival analysis.

1 Introduction

Very often *Open-Source Software* (*OSS*) is developed by decentralized teams of programmers, who cooperate globally using web-based source code repositories. There are several features typically associated with such *Collaborative Innovation Networks* (*COINs*): (a) voluntary work; (b) low organizational costs; (c) meritocracy. In recent years COINs have proved their ability to produce high quality software. Leading example of such network is the *GitHub website*, an

L.M. Aiello and D. McFarland (Eds.): SocInfo 2014, LNCS 8851, pp. 80–94, 2014.

online social network that supports development of software by virtual teams of programmers. Every *GitHub user* can create his/her own repository and work on it with other registered users. They may also join projects created by somebody else and make their own contributions there. *GitHub* has no recommendation system for developers, which would support their decisions on how to contribute effectively from a scratch. Every GitHub user makes his/her own decision on how to manage their personal time and professional skills - that is the reason why the process of *team formation* on GitHub is decentralized and might be considered as self-organizing. In other words, there is no central mechanism governing the formation of OSS teams.

It sheds light on the puzzling fact that even though open source software (OSS) constitutes public good, it is being developed for free by highly qualified, young, motivated individuals, and evolves at a rapid pace. We show that when OSS development is understood as the private provision of public good, these features emerge quite naturally. We adapt a dynamic private provision of public goods model to reflect key aspects of the OSS phenomenon, such as play value or homo ludens payoff, user-programmers' and gift culture benefits. Such intrinsic motives feature extensively in the wider OSS literature and contribute new insights to the economic analysis

1.1 Problem Definition

Foregoing facts awaken our interest in investigating if there are any significant correlations between *project quality* and characteristics of the *team members*. We consider project quality as consisting of two aspects: one is the *number of stars* that any project might receive from GitHub users, and the second one is the response of project team to issues reported for a given repository. In case of the first indicator, we believe that community reaction to the project is a proper measure of its quality. Any GitHub user is able to gratitude a chosen projects with a *star* – it shows his admiration and positive attitude towards chosen repository.

It is very important for every kind of software to have a good support - that is a team of people, who are able to respond, in case when the community of users reports some *bugs* and *feature requests*. Considering any piece of software, bugs constitutes an almost inevitable part of its lifetime – even alpha and beta tests are not able to rule out all possible problems with the software. Moreover, community of users is the best source of information about the performance of solutions that have been implemented and about the lack of some features, which might significantly improve usability of the created system. Open-Source Software developed on GitHub by COINs is no exception here – it also needs maintenance of issues reported by community of users.

Github platform has a distinct functionality for reporting issues on a project: it allows GitHub users to report such things as bugs, feature requests or enhancements to the team of developers. The categories of all possible issues might be defined by the owner of the repository. For each repository from GitHub we have a record of its issues survival - data about moment when a particular issue had

been opened, and eventually when it was closed. The analysis of survival of those issues gives us insights about typical life duration of issues in different kinds of GitHub projects.

We believe that *issue survival* is one of the indicators of GitHub *team quality*. Our assumption is, that well organized and motivated teams tend to maintain and swiftly close issues associated with their repositories, and measuring the time of issue closure, together with other *predictor variables* describing GitHub repositories, is a good metric of quality for the team maintaining a given repository.

We have also gathered several characteristics of GitHub projects and analyzed their influence on project popularity and the quality of support offered to users. Several interesting conclusions were made, for example, it is better to attract focused active developers to the project than to attract popular members.

2 Related Work

Questions concerning the problem of *quality* in *Open-Source Software* (OSS) and *COIN*s has been investigated by several researchers. A generic review of the empirical research on Free/Libre and Open-Source Software (*FLOSS*) development and assessment of the state of the literature might be found in ACM article by Crowston et.al. (2008) *"Free/Libre Open-source Software Development: What We Know and What We Do Not Know"*.[2]. In publication *"Software Product Quality Models"* by Ferenc et.al. (2014) authors provide a brief overview about the history of software product quality measurement, focusing on software maintainability, and the existing approaches and high-level models for characterizing software product quality. Based on objective aspects, the implementations of the most popular software maintainability models are compared and evaluated. This paper also presents the result of comparing the features and stability of the tools and the different models on a large number of open-source Java projects.[5] However, we have not found many papers that directly investigate relations between projects issues survival and its quality.

A solid understanding of online collaboration is provided by research on wikiteams. Wikipedia is a laboratory for open, virtual teamwork.[17] It is also a community similar to GitHub, because collaboration manifests through a swarm creativity which is a part of COIN model. Scholars Hupa et.al. (2010) enhance expert matchmaking and recommender systems with multidimensional social networks (MDSN).[8] According to them, dimensions of: trust, acquaintance and knowledge store information about the social context of an individual, as well as team's social capital, intra-group trust and skill difference. Social network is based on the entire Wikipedia edit history, and therefore is a summary of all recorded author interactions.[18] Using information from these dimensions they define a criteria that predict team performance.[8] A dimension of distrust is added to model because of its beneficial behaviour to teams quality.[19]

Rahmani, Khazanchi (2010) published *"A Study on Defect Density of Open Source Software"*, where they present an empirical study of the relationship between defect density and download number, software size and developer number

as three popular repository metrics. Contrary to theoretical expectations, their regression analysis discovered no significant relationship between defect density and number of downloads in OSS projects. Yet, researcher find that the number of developers and software project size together present some promise in explaining defect density of OSS projects. They plan to explore other potential predictors for defect density in OSS projects, together with the use of non-linear regression to explain the trends in defect density associated with OSS project.[16]

Michlmayr, Senyard (2006) in their paper *"A Statistical Analysis of Defects in Debian and Strategies for Improving Quality in Free Software Projects"* analyse 7000 tickets from the Debian issue tracking system. This data accumulated during over 2.5 years allowed them to make conclusions regarding a high-maturity project through analysing their issues closure. Scholars found that the number of issues is increasing together with the decrease in a defect removal rate. Scientist found that frequent releases lead to shorter defect removal times and possibly to more user feedback. Secondly, they argued that a close interaction with the upstream authors of free software is beneficial, and upstream authors gain from wide testing and more user feedback. Finally, working in groups increases the reliability of volunteer maintainers and leads to shorter defect removal times.[13]

Related to our approach is work by Fischer et.al. (2003) *"Analyzing and relating bug report data for feature tracking""*, where bug reports tracks were used to investigate software evolution. Authors method has been validated using the large open source software project of *Mozilla* and its bug reporting system *Bugzilla*. Their approach uncovers hidden relationships between features via problem report analysis and presents them in easy to evaluate visual form.[6]

As one can observe, there is a lot of research concerning quality in Open-Source Software, and their number happens to grow after the success of the *SourceForge* and *GitHub* portal. Internet databanks, which aggregate data from different web-based online source repositories, make for the analysis of Open-Source Software easier and wider. Researchers Farah et.al. (2014) published work titled *"OpenHub: A scalable architecture for the analysis of software quality attributes"* where they analyze $140,000$ *Python* repositories under quality attributes - performance, testability, usability, maintainability. Scientists merged information on Python repositories collected from GitHub with metrics generated by OpenHub (formerly Ohloh) - an internet aggregator for OSS repositories.[4]

Interesting analysis of activity fade-out in OSS projects are presented in *"Is it all lost? A study of inactive open source projects"* by Khondhu et.al. (2013). Researchers quote an informal rule, according to which "when developers lose interest in their project, their last duty is to hand it off to a competent successor". However, mechanism of such hand-off is not widely known among OSS users. Paper goal is to differentiate projects that had maintainability issues from those that were inactive for other reasons.[11]

A discussion about central management vs. OSS is covered in book by O'Reilly Media *"Making Software: What Really Works, and Why We Believe It"*.[15] Mahony, Ferraro (2007) prove that successful communities structure their work and that good communities and teams are self-governing.[14] There also have

been attempts to support programmers work, by recommendation engines. In paper from Zimmermann et.al. (2014) *"Mining version histories to guide software changes"* data mining has been applied to version histories, in order to guide programmers along the related changes. Their system prototype called ROSE was able to correctly predict 26% of further files to be changed—and 15% of the precise functions or variables.[20]

In work of Kalliamvakou et.al. (2014) researchers indicate that, although GitHub is a rich source of data, there are also potential perils that should be taken into consideration. Among other things they show that the majority of the projects on GitHub are personal and inactive. According to their research two thirds of projects (71.6% of repositories) are personal – the number of committers per project is very skewed: 67% of projects have only one committer, 87% have two or less, and 93% three or less. This findings shows that most of the users do not use GitHub for collaboration on projects. However, in our studies we have been focused on most popular repositories, which eliminates one-person projects[10].

Finally, different case studies of chosen GitHub repositories reveal even more interesting conclusions. In paper *"Social coding in GitHub: transparency and collaboration in an open software repository"* by Dabbish et.al. (2012) a series of in-depth interviews with central and peripheral GitHub users was performed. Authors claim that people make a surprisingly rich set of social inferences from the networked activity information in GitHub, such as inferring someone else's technical goals and vision when they edit code, or guessing which of several similar projects has the best chance of thriving in the long term. It is suggested that users combine these inferences into effective strategies for coordinating work, advancing technical skills and managing their reputation.[3] Another series of interviews with GitHub developers reader might be found in *"Performance and participation in open source software on GitHub"* McDonald et.al. (2013). Authors conducted qualitative, research with lead and core developers on three successful projects on GitHub. They aim was to understand how OSS communities on GitHub measure success. Two main findings were reported: first, lead and core members of the projects display a nuanced understanding of community participation in their assessment of success; second, they attribute increased participation on their projects to the features and usability provided by GitHub.[12]

3 Dataset

3.1 Predictor Variables

We now describe the dataset containing representative GitHub projects used in this paper. We used *Google BigQuery* online tool to create a list of GitHub repositories (or 'repos' for short), sorted descending by their highest peak in trend (received attention from Internet users) during a month. We define trend by the biggest increase in popularity during a month. Popularity of a repository is measured by its number of stars (number of 'stargazers'). We analysed mature repositories existing for at least two years. There are together 164418 GitHub

repositories on the list. This list simply consists of repositories, but lacks information on their team members (contributors and collaborators). From this big set of repositories we selected 2000 of them with the highest increase in popularity during a month. In this way we avoid taking into consideration projects which are personal or inactive – inactive repositories were one of the main perils of GitHub data described in Kalliamvakou et.al. (2014)[10].

Each record in our dataset has 12 columns, which are: 'repository owner', 'repository url', 'repository name', 'biggest increase in popularity', 'repository description', 'is a fork', 'wiki enabled', 'when pushed at', 'master branch', 'issues enabled', 'downloads enabled', 'repository creation date'. Additionally, we also have information on below values on any moment of time during the repository existence: 'number of stars', 'number of forks', 'number of pushes', 'how many issues open/closed etc.

Next step is to receive information on developers in those entry teams, which we call x-axis attributes for a repository. For this purpose, we use *GitHub API* to parse missing information. For the mentioned 2000 repositories we downloaded through a script (available freely here: `https://github.com/wikiteams/supra-repos-x`) below additional information on a developer: 'developer username (login)', 'developer name', 'developer followers count', 'developer following count', 'developer company', 'number of repos developer contributed to', 'number of repos he owns', 'date when developer registered', 'developer location', 'is developer hireable', 'is developer working during business hours', 'developer typical working period', 'gists count', 'private repos count'. Also, more properties for a repository (*y-axis*) are downloaded: 'repository default branch', 'opened issues count', 'repository organization', 'repository language (main technology)'.

Good source of general developer activity on GitHub is a data source called *OSRC* report card, from where we download aggregated data regarding the *user activity time*. We calculate two additional attributes. Firstly, we want to check whether the developer contributes mostly during working hours (between 9 and 17 o'clock) in his local time, or he is an active GitHub user but committing beyond this period of time. Secondly, we calculate a working period (in hours) for this developer. We define a working period as a sum of hours in the biggest rectangle drawn on the daily activity histogram.

3.2 Repositories Issues

Any change in an *Issue* is recorded in a databank called *GitHub Archive* (in short - 'GHA'). It is a third party project to record the public *GitHub timeline* and make it easily accessible for further analysis. In GHA, every time when some issue is opened, closed or reopened, 'IssuesEvent' is stored to a database. Firstly, we downloaded all data collected in year 2013 from the *GitHub Archive*. Secondly, we selected IssuesEvents to create a history of issue creation on all *GitHub repositories* during that year. IssuesEvent is triggered whenever an issue is created, closed or reopened, and the GHA collects those events.

Data was merged into full information record on each single issue. It contained opening and closing date of the issue and a calculated difference: the time span. Once created, the issue in a GitHub repository cannot be deleted, it can be only closed. Finally, we used the *GitHub API* to query for issue labels, which GHA didn't provide.

We managed to create a dataset of issues with following attributes: repository owner (a person or organization who manages the code repository as a privileged user), repository url (an 1-1 identifying repository key, a web address which allows to view the repository in a browser), repository name, issue number, issue status (opened or closed), 'opened at' (when was the issue created), 'closed at' (when was the issue last time resolved), difference in minutes (also hours and days, difference between fields 'opened at' and 'closed at').

3.3 Data Preprocessing

Since we are drawing conclusions on the whole projects, not individual developers, characteristics of project members had to be aggregated into single attributes. Here we simply computed means of each attribute over all project members. Such attributes are prefixed by 'average.' (alias 'avg.').

Many attributes exhibited highly skewed, power-law like distributions, which are difficult to model with statistical methods. Logarithmic transformation $x' = \log_{10}(x + 10)$ has been applied to the following attributes to decrease the skew:

'forks_count', 'network_count', 'average.developer_followers',
'average.developer_following', 'average.developer_contributions',
'average.developer_total_public_repos', 'average.developers_works_period',
'average.public_gists', 'commits_count', 'branches_count', 'releases_count',
'contributors_count'.

4 Measures of Project's Quality

In order to discover factors influencing project quality we need to be able to precisely measure project quality. Unfortunately, the task is not easy, as there are many possible criteria, which are not always easy neither to measure nor to evaluate. In this chapter, we are going to introduce and describe two GitHub project *quality metrics*, based on project popularity and the quality of user support offered by team members.

4.1 Attractiveness and Popularity – Stargazers

The first metric we analyze is the number of *stars* the project has, i.e. how many times it has been endorsed by members of the GitHub community. For each project, we gathered the number of stargazers - users who starred a given project.

Since the stargazers count follows a *power-law distribution* (it means there are lots of projects with few stars and a few projects with a very large number

of stars), it is not suitable for e.g. regression analysis. We applied logarithmic transformation $x' = \log_{10}(x + 10)$ before using it as a metric of project quality. The resulting quantity has a well behaved distribution as can be seen on its histogram shown in Figure no. 1. The offset 10 is provided to avoid taking logarithms of zero and to reduce skew for small values.

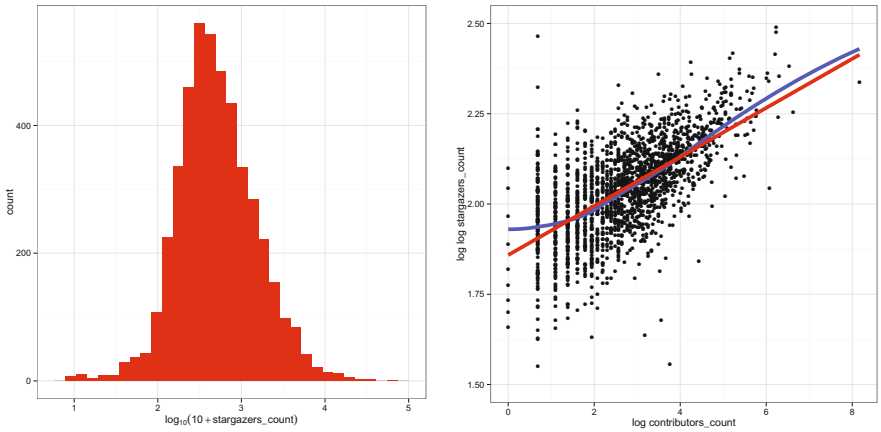

Fig. 1. Left - histogram of the number of stargazers for different project. Right - dependence of the number of stargazers on the number of contributors to a project.

The metric may than be used to analyze the factors influencing project quality. As an example of the type of analysis for which it can be used we show the dependence of the metric on the logarithm of the number of contributors to the project, see the right part of Figure no. 1.

The red line shows the linear regression fit and the blue line a nonlinear LOESS regression fit [1]. Clearly, the larger the number of contributors is, the larger the number of stars. This by itself is not surprising; however, what is interesting is that the number of stars grows exponentially with the number of contributors. Indeed, the nonlinear fit is almost identical to the linear one (recall that we use a double logarithm of the number of stars). It is not yet clear to us whether it is the project's popularity that attracts contributors or, vice-versa, the large number of contributors results in good and, consequently, popular projects.

4.2 Quality of Support – Survival of Issues

We now describe the second metric of a project quality introduced in this paper. It is based on the time it takes the project team members to close issues related to the project. From now on, we will use the tools of *survival analysis*.

Survival analysis focuses typically on times to a given event – it might be loss of some user, customer migration, or death in case of biological research. One of the typical questions, which survival analysis attempts to answer, is: what is the proportion of a population which survived a certain amount of time? Of course in case of *GitHub issues* our question is - what proportion of issues was not closed before some particular point in time.

A key aspect of survival analysis [9] is censoring - if an issue was opened just a month ago, at the current time point we do not know, whether it will be closed within a year or not. In order to handle censoring in a statistically proper way we use the *Kaplan-Meier estimates* of survival time for issues of a given project. The left part of Figure no. 2 shows the survival curve for issues of an example project. It can be seen that a certain percentage of issues is closed very rapidly, indicating active support of users by the project's team. However, older issues are often not closed at all. In total, about 50% of issues is not being addressed, suggesting that there is a high chance, that user problems will not be resolved. The '+' marks on the curve indicate the age of issues opened recently, which have not yet been closed and have not reached the maximum time displayed on the x axis. The right part of the figure shows the combined survival curve for issues of all analyzed GitHub projects. It can be seen that the response times are usually fast, but many issues have not been addressed at all.

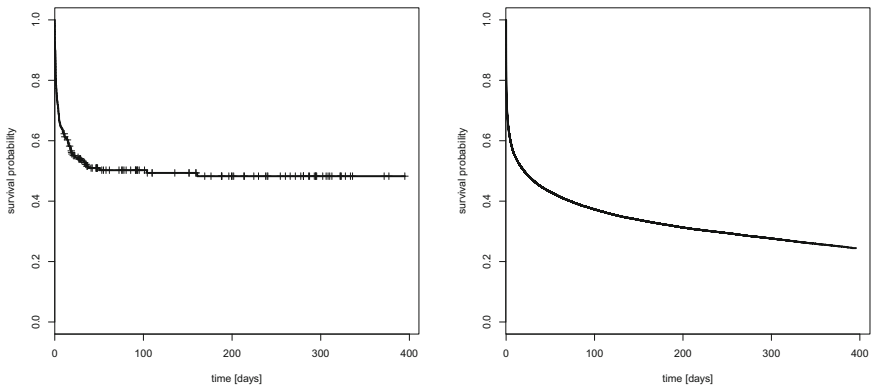

Fig. 2. Survival curves for issues of the GateOne project and for all projects combined

In order to facilitate the assessment of project quality, we devised summary metric for survival curves. To this end, we computed survival probabilities for issues after 1, 2, 3, 7, 30, 100, and 365 days. Performing the PCA *(Principal Components Analysis* on those probabilities revealed that just two components are enough to explain 96% of the variance of the seven probabilities. Further, the first component describes (roughly) the average of the percentages of bugs closed after different amounts of time, and the second one differentiates the probabilities of issues being closed rapidly, in a matter of days.

Consequently, we decided to summarize the survival curve for each project with just two numbers: the percentage of issues closed after 3 and 365 days. It turned out that those two numbers explain about 94% of the total variability of the seven issue survival probabilities—almost the same as the first two principal components—while being much easier to understand.

To summarize, we have provided the concise metric of GitHub project quality based on the time needed to respond to the issues. The metric is based on two numbers measuring how quickly the user may expect a response and what are the chances that his or her issue will be eventually resolved.

5 What Makes a Good GitHub Project?

In this section we are going to analyze how different characteristics of a project and its developers influence the two aspects of its project quality.

5.1 What Affects the Project Popularity?

We have conducted a regression analysis to discover factors influencing project popularity. Since the programming language specified for the project is a categorical attribute with many (55) values, we decided to exclude it from the initial analysis and analyze it separately in the next chapter (no. 5.2). Only the information whether the project's language was specified is included.

Application of linear regression resulted in a model with a very high multiple R^2 coefficient of 0.779. In other words, almost 80% of the variability of projects popularity (after the logarithmic transform) is explained by project features. The most significant variable in the model was `forks_count` - the number of times the fork of this project was created. The number of forks reflects general amount of activity in the project, so it is a logical conclusion, that active projects are more popular. Few other attributes turned out to be highly correlated with the number of forks and they also reflect general amount of project activity and a project size. Those attributes are commit_count, contributor_count, releases_count, branches_count and network_count. Another pair of mutually correlated attributes, repo.updated_at and repo.pushed_at, also reflects the amount of activity in the project.

Unfortunately, those correlations are of little practical use, since project activity is likely an *effect* of its popularity (developers are more likely to fork a project, if it is well known and attractive), not necessarily its cause, at least not the primary one. Those attributes have thus been removed from the dataset before further analysis.

A new regression model was built on the reduced dataset. Variables with statistically significance (p-value below 0.05) coefficients are shown in Table no. 1. The first column gives us the attribute's name, the second its coefficient in the regression model, and the third one is the p-value indicating statistical significance of the coefficient. Higher values of coefficients mean that a given quantity has a positive influence on project's popularity. Since the number of stars has

been logarithmized, the coefficients should be interpreted multiplicatively. For example, the coefficient for repository creation time, -0.144 means that an increase in creation time by one year corresponds to the number of stars increasing $10^{-0.144} = 0.72$ times (in fact decreasing).

It can be seen, that the most significant attribute is project creation time. The coefficient is negative, so projects created later have, in general, less stars. This is obvious since, the stars accumulate over time giving older projects a natural advantage. A similar attribute is the average time at which developer accounts were created. Surprisingly this time the relation is opposite, having newer developers in the project is positively correlated with its popularity. This phenomenon can probably be explained by the fact that programmers may join Github in order to contribute to attractive projects.

Another observation is, that forks of other projects are in general much less popular. This is plausible since forks can be created very easily on Github and are often used as a part of the development process, not necessarily constituting separate projects. The language of the project being specified is correlated with popularity (see a dedicated section below for a discussion).

Another significant attribute is whether the project is owned by an organization. Projects owned by companies and other organizations are in general more popular.

Another two significant attributes: the average number of followers of developers in the project and the average number of developers/projects followed by them can be seen as an approximation of social relations of project members. Actually, it turns out that project whose developers follow many others are in general more popular. Surprisingly the effect for developers being followed by many others (i.e. having popular developers in the project) is much weaker. This discovery results in a practical advice for projects: finding developers engaged in the community is good for the project popularity and can be measured by a simple proxy quantity.

Another two related attributes are the average number of repositories owned by project members and the total amount of their contributions (including other projects). The coefficients here are negative offering another practical advice to project managers: try getting people who will be able to concentrate on your project without spreading attention on too many other projects.

5.2 Programming Language

We will now analyse how the project's programming language is related to its popularity. To discover this, we have built a regression model based on just one attribute, *repository language*. As previously mentioned, there are 54 programming languages used in the analyzed projects, plus an extra value for no language specified.

It turns out that the programming language has little effect on the projects popularity, with a few exceptions - significant influence was observed for only 4 cases. The most significant effect was that projects written in *Common Lisp* are

Table 1. Regression model

attribute	coefficient	p-value
repo.created_at	-0.144	$p < 2.0 \cdot 10^{-16}$
is_fork	-0.272	$p = 2.2 \cdot 10^{-6}$
language.specified	0.290	$p = 8.8 \cdot 10^{-7}$
organization.specified	0.163	$p = 5.0 \cdot 10^{-12}$
average.developer_followers	0.057	$p = 0.029$
average.developer_following	0.174	$p = 0.002$
average.developer_contributions	-0.094	$p = 0.009$
average.developer_created_at	0.120	$p = 2.8 \cdot 10^{-11}$
average.developer_total_public_repos	-0.163	$p = 0.042$
average.developers_works_period	0.108	$p = 5.3 \cdot 10^{-6}$
Observations	1755	
R^2	0.203	

less popular. The coefficient was -0.702, significant with p-value of $2.3 \cdot 10^{-3}$. After taking into account the log-transformation of the number of stars, this roughly translates to those projects having 5 times less stars than similar projects written in other languages. The probable explanation is that Common Lisp is an old technology, nowadays used only by a small fraction of developers for very specialized purposes.

Another significant fact was that projects that did not specify the programming language were also significantly less popular. An inspection revealed that many of those projects include color themes, documentation etc. which may not be very popular among users. Moreover, GitHub assigns project language automatically, based on file contents, so projects with no particular files are assigned to this category.

On the contrary, projects based on CSS styles were more likely to be popular, probably due to the growing popularity of web-based technologies.

5.3 What Affects the Quality of a Support?

We now move on to the analysis of the quality of projects from a short and long term support. Since the survival probabilities are not normally distributed, we have used *binomial regression* (a variant of logistic regression) to model it. Each data record has a number of trials n and a number of successes n_1 assigned. A generalized linear model is then built, which predicts the probability of success p, assuming that n_1 follows, in each record, the binomial distribution with parameters n and p (for more details, see *Hosmer, Lemeshow* book [7]). In our case n corresponds to the total number of project's issues, p to the estimated probability of bug survival. We set the n_1 to

$$n_1 = n \cdot p_t, t \in \{3, 365\}$$

where p_t is the *Kaplan-Meier* estimate of the fraction of surviving issues defined in the previous section. Hence, that n_1 needs to have non-integer values - however, this is not a problem for the implementation of logistic regression available in the R statistical package.

We begin by analyzing the influence of attributes related to project size and general activity. For both - short term (3 days bug survival) and long term (365 days bug survival) support - the most important attribute is the number of branches, which is negatively correlated with bug survival. This means that a large number of branches has a positive influence on the project. Since a typical *git workflow* for fixing a bug involves creating a branch, making the changes and merging the branch back, this correlation is logical.

Unfortunately, the number of branches is correlated with general project activity and thus, as was the case with project popularity, we removed this and correlated attributes before further analysis. Those attributes are commit_count, contributor_count, releases_count, branches_count and network_count. Another pair of mutually correlated attributes, repo.updated_at and repo.pushed_at.

The model was then rebuilt. Table no. 2 shows the significant regression coefficients for both short and long term bug survival. Note that negative values of the coefficients are desired here as they translate to lower numbers of surviving bugs.

Table 2. Regression coefficients for short and long term bug survival

3 day bug survival		
attribute	coefficient	p-value
repo.created_at	-0.014	$p = 0.049$
is_fork	-0.926	$p = 0.017$
has_downloads	-0.055	$p = 0.021$
average.developer_following	2.158	$p = 0.002$
average.developer_contributions	1.787	$p < 2.0 \cdot 10^{-16}$
average.developer_hireable	-1.598	$p = 5.7 \cdot 10^{-7}$
average.developer_total_public_repos	0.235	$p = 0.009$
average.developers_works_period	0.314	$p = 0.004$
365 day bug survival		
attribute	coefficient	p-value
organization.specified	0.079	$p = 0.005$
average.dev_name_given	-0.182	$p = 0.031$
average.developer_following	-1.194	$p = 0.001$
average.developer_contributions	1.337	$p < 2.0 \cdot 10^{-16}$
average.developer_hireable	2.317	$p = 0.002$
average.developer_works_during_bd	0.329	$p = 0.029$
average.public_gists	0.391	$p = 0.022$

Let us now comment on the significant attributes. First of all, it can be seen that having developers making many contributions (including other projects) and owning many repositories negatively influences the number of bugs fixed.

The same advice can be offered as in the case of project popularity - try getting into the project focused developers who will concentrate all their efforts on it.

The number of projects/developers followed by team members is again an important factor. However, surprisingly, it has a positive effect only on fixing bugs in long term, not on 'rapid response' to user issues. This issue needs to be investigated further.

The effect of projects being run by employed developers (attributes 'organization.specified' and 'average.developer_works_during_bd') is significantly negative towards addressing longstanding bugs. The probable reason is that organizations are unwilling to commit resources to fixing user issues and prefer to concentrate on aspects of the project which are important to them.

6 Conclusions and Future Research

Our paper presented *two measures of quality* for GitHub Open-Source Software projects. One is based on a project popularity, the other one is based on how fast the project's team solves issues reported by users. We have also collected several attributes describing projects and their developers and analyzed their influence on those quality measures. Together, it resulted in making several interesting discoveries. For example, it is better for a software project to have focused developers involved in the community rather than having in the team popular, often followed developers. Future work will focus on detailed studies of what aspects of a team collaboration affect a project quality.

Acknowledgements. This work is supported by Polish National Science Centre grant 2012/05/B/ST6/03364

References

1. Cleveland, W.S., Devlin, S.J.: Journal of the American Statistical Association 83(403), 596–610 (1988)
2. Crowston, K., Wei, K., Howison, J., Wiggins, A.: Free/libre open-source software development: What we know and what we do not know. ACM Comput. Surv. 44(2), 7:1–7:35 (2008)
3. Dabbish, L., Stuart, C., Tsay, J., Herbsleb, J.: Social coding in github: Transparency and collaboration in an open software repository. In: Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work, CSCW 2012, pp. 1277–1286. ACM, New York (2012)
4. Farah, G., Tejada, J.S., Correal, D.: Openhub: a scalable architecture for the analysis of software quality attributes. In: Proceedings of the 11th Working Conference on Mining Software Repositories, pp. 420–423. ACM (2014)
5. Ferenc, R., Hegedus, P., Gyimothy, T.: Software product quality models. In: Mens, T., Serebrenik, A., Cleve, A. (eds.) Evolving Software Systems, pp. 65–100. Springer, Heidelberg (2014)
6. Fischer, M., Pinzger, M., Gall, H.: Analyzing and relating bug report data for feature tracking. In: 2013 20th Working Conference on Reverse Engineering (WCRE), p. 90. IEEE Computer Society (2003)

7. Hosmer, D.W., Lemeshow, S.: Applied logistic regression. Wiley-Interscience Publication (2000)
8. Hupa, A., Rzadca, K., Wierzbicki, A., Datta, A.: Interdisciplinary matchmaking: Choosing collaborators by skill, acquaintance and trust. In: Abraham, A., Hassanien, A., Snášel, V. (eds.) Computational Social Network Analysis. Computer Communications and Networks, pp. 319–347. Springer, London (2010)
9. Kalbfleisch, J.D., Prentice, R.L.: The statistical analysis of failure time data. John Wiley & Sons (2002)
10. Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D.M., Damian, D.: The promises and perils of mining github. In: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, pp. 92–101. ACM, New York (2014)
11. Khondhu, J., Capiluppi, A., Stol, K.-J.: Is it all lost? a study of inactive open source projects. In: Open Source Software: Quality Verification, pp. 61–79. Springer (2013)
12. McDonald, N., Goggins, S.: Performance and participation in open source software on github. In: CHI 2013 Extended Abstracts on Human Factors in Computing Systems, CHI EA 2013, pp. 139–144. ACM, New York (2013)
13. Michlmayr, M., Senyard, A.: A statistical analysis of defects in debian and strategies for improving quality in free software projects. The Economics of Open Source Software Development, 131–148 (2006)
14. O'Mahony, S., Ferraro, F.: The emergence of governance in an open source community. Academy of Management Journal 50(5), 1079–1106 (2007)
15. Oram, A., Wilson, G.: Making Software: What Really Works, and Why We Believe It. O'Reilly Media (2010)
16. Rahmani, C., Khazanchi, D.: A study on defect density of open source software. In: 2010 IEEE/ACIS 9th International Conference on Computer and Information Science (ICIS), pp. 679–683. IEEE (2010)
17. Turek, P.: Wikiteams: How do they achieve success? IEEE Potentials 30(5), 15–20 (September 2011)
18. Turek, P., Wierzbicki, A., Nielek, R., Hupa, A., Datta, A.: Learning about the quality of teamwork from wikiteams. In: 2010 IEEE Second International Conference on Social Computing (SocialCom), pp. 17–24 (August 2010)
19. Wierzbicki, A., Turek, P., Nielek, R.: Learning about team collaboration from wikipedia edit history. In: Proceedings of the 6th International Symposium on Wikis and Open Collaboration, WikiSym 2010, pp. 27:1–27:2. ACM, New York (2010)
20. Zimmermann, T., Weissgerber, P.: Mining version histories to guide software changes. In: 26th International Conference on Software Engineering (ICSE 2004), pp. 563–572 (2004)