The 29th Annual Conference of the Japanese Society for Artificial Intelligence, 2015

1L5-5

Activity Dependency in Collaborative Networks

Pablo Loyola Yutaka Matsuo

Graduate School of Engineering, The University of Tokyo

Open Source Software (OSS), as well as other distributed collaborative initiatives such as Wikipedia, is based on a continuous stream of voluntary contributions from developers with different incentives, skills and motivations. In this work, we focus on analyzing the behavioral patterns that emerge after a contribution is merged to the official codebase. In particular, we are interested in the activities generated by the community after they assimilate a particular contribution. This subsequent activity usually involves patches, related contributions and bug fixes, which arrive in the form of a chain reaction, especially when it is generated from a highly reputed member. We studied this phenomenon taking into account both technical and social aspects of the software development process and generated a prediction model based on the concept of cascades. Our initial experiments shows that this approach has the potential to simulate real contribution dynamics.

1. Introduction

In Open Source Software (OSS) development dynamics, the stream of continuous contributions represents the key aspect that maintains the project alive [8]. These contributions are not only additions of new features, but also bug fixes, discussions and test suites.

The main characteristics of this phenomenon, such as the non-centralized development and the heterogeneity of the participants, have several advantages [10, 9], but also represent a challenging factor in relation to the managerial aspects of the software engineering process. A considerable proportion of OSS projects follows a *pull request* contribution model [5], where it is necessary to review and discuss every new contribution before it is merged to the official branch of development. As the people in charge of reviewing the contributions is scarce (usually core members), it could be convenient to have in advance an estimation of the contribution activities in order to plan the distribution of workload in an effective way.

Therefore, the remaining question is how to generate a prediction model for the contribution activity. Although standard methodologies for prediction can be used, we hypothesize that their inherently shallow structure will not incorporate the intrinsic characteristics of the contribution dynamics. We believe that the level of activity at a certain point should not be predicted using only data from a predefined *initial condition*. Therefore, the intermediate behaviors should be considered. The main rationale for that assumption is that there is a dependency between the activities in the sense that a given contribution ignites a set of related contributions. An example is presented in Figure 1: user A submitted a contribution a, which was merged to the official branch. Contribution a was the addition of a new feature to the system. User B analyzed this contribution and decides to build something else on top of a. Therefore, she submitted a contribution b, which was reviewed and merged to the official branch. Subsequently, user C detected a compatibility issue related to b which was not seen during the code reviewing process, therefore, she submitted a bug fix c.

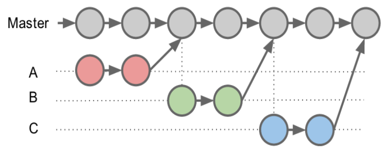

Figure 1: Example of dependency in contribution dynamics.

As we can see from the previous example, one particular contribution triggers the generation of set of related contributions, which in turn trigger another set, conforming a *chain reaction*, until the activity naturally decays or reach and steady state. Initial evidence shows that this type of behavior occurs and it common when the the project hits a milestone, such as the release of a new version, or when a highly reputed member submits a contribution [4].

We took as a main inspiration the work conducted in the field of social network analysis, specifically on the cascade phenomenon, which is the diffusion of content in a network through a *re-sharing* functionality. The rationale for this is that the problem we are considering shares the same nature with the content cascades in social networks: We are in presence of a initial action that triggers a sequence of related activities.

Our approach consists of firstly identifying the groups of linked activities (along the initial igniting contribution), then we extract a set of features that allow to characterize each group. Subsequently, we define the prediction problem as a binary classification: given that the post contribution activity has reached a length of k, we would like to predict if it will reach $m(k)$, which the median length of all groups of linked activities that have at least k elements.

We performed an empirical evaluation using data from the Github collaborative platform, which allowed to extract both technical and social data from the development process and performed a exploratory study. Our initial results shows that the proposed approach has the potential to predict dependency length in a reliable way.

Contact: Pablo Loyola, pablo@weblab.t.u-tokyo.ac.jp

2. Proposed Approach

Our main inspiration comes from the study of content dissemination in Social Networks, where the concept of *cascade* is defined as the process in which a piece of content is shared sequentially among participants through a *re-sharing* mechanism.

Our observation on OSS activity allowed us to realize that the contribution dynamics have certain similarities with the cascade phenomenon. As the development of an artifact (a new feature or a bug fix) is based on a collaborative process, each new component is built on top of the previous work. Therefore, there is a chain of dependency between each contribution and the later activity generated. As source code is accessible for all the group of developers and subversion systems, such as Git, provide a robust and clean handling of all the activity, each participant is able to assimilate the latest changes in an easy way. Then, each new contribution opens a new space of possibilities for the rest of the team: Some of them, could visualize a new feature, given the new set of instructions added. Others may find a backward compatibility issue, which will need to be fixed. Literature shows examples of dependent behavior in contributions for several OSS projects, in the form of *herding*, as reported in [4].

Therefore, one particular action triggers a group of linked activities, generating a *cascade* of contributions. We are interested in studying the dynamics of this group of activities by predicting its length. This means, given an initial contribution, we would like to estimate how many related activities will be generated.

To achieve such goal, we took the state of the art on cascade prediction, a recent work by Cheng et al. [3], as a methodological framework to build our approach. In their paper, the authors modeled the cascade prediction problem not as as the estimation of the final length given initial conditions, but as a continuous ensemble of sub-predictions. In this sense, given a cascade has reached a length of k, it is desired to predict if it will grow beyond the median of sizes $f(k)$, $f(k) > k$. Therefore, the task is composed by a sequence of predictions, one for each observed size. This configuration has several advantages, such as that classes will eventually follow a more homogeneous. But in the context of designing a tool that can support the management of collaborative tasks such as OSS development, the ability to monitor the performance and provide an estimation of the upcoming activity in a continuous way, is the one that appears more relevant.

Contrary to the Social Network scenario, where cascades are explicitly available, in OSS it is necessary to initially find a reliable way to group activities in a way that they represent a sequential and dependent chain of events. Once the cascades are identified, we define the prediction problem and proposed a method for obtaining the corresponding estimation.

2.1 Model

We assume a standard decentralized collaborative environment in which a set of U users work under a semi-guided way on a artifact A that is conformed by a set of E elements. Each user $u_i \in U$ can access anytime and visualize the the state of each component $e_j \in E$. Examples of this can be found in OSS development, Wikipedia and any other collaborative content generation.

Actions: An action is part of the set of feasible procedures that can be applied on any $e_j \in E$. For example, in an OSS context, an action a is such that $a \in \{create, update, delete\}$, as this set represent what a user can do.

Contribution: A *contribution* is defined as a set of actions carried out by a user $u_i \in U$ over a subset of k elements of A in a time t, namely $C_{iKt} = \{(u_i, a_j, c_1, t), (u_i, a_j, c_2, t), ..., (u_i, a_j, c_K, t)\}$. This representation is usually present in OSS development ecosystems, as they usually relies on a subversion systems such as Git, where the user work locally and then submits his contribution for revision in the form of a *pull request.* Therefore, all the actions are encapsulated into a defined set which has associated a time-stamp and a user.

Contribution Cascade: A contribution cascade is a sequence of contributions that are linked on a dependency basis. The cascade begins by an initial contribution which modifies the artifact under development in a specific way and that is carried out by a particular user. We called this contribution the *initial contribution.* The changes produced by the initial contribution on the artifact are noticed and acknowledged by the rest of the group of users, which in turn may decide to generate a new contribution based on it. Therefore the following contribution depends on the current one.

2.2 Contribution Cascade Identification

We need to group the contributions in a way that takes into account the similarity of the software components modified but at the same time the dependency in terms of time.

Grouping software components is a task already studied by the research community. Several methods have been proposed such as [11, 2]. The dependency between components have also been studied and it is commonly known as *coupling* [1]. But these methods are mainly focused on grouping software artifacts, such as files and libraries, based on co-change frequency. In our case, we are interested in the inverse task, as it is necessary to group the activities conducted on the set of artifacts.

While the natural idea is to group together contributions that change similar sets of components, in the case of software development the explicit dependency between the modules that are being modified needs to be taken into consideration. To illustrate this issue, Figure 2 shows the feasible configurations between two given contributions (namely *red* and *blue*), assumed coming from different users at different times ($t_{red} < t_{blue}$). In (a) it can be seen that both contributions are focused on disjoint sets of modules, but there is an implicit relationship between them, as the software components they modify are dependent, therefore we call this configuration *(non-overlapping, dependent).* In (b), we have some shared elements among contributions, which are also dependent, therefore, this is a *(overlapping, dependent)* configuration. Similarly, we have *(overlapping,*

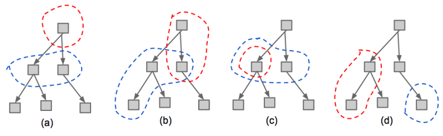

Figure 2: Instances of overlapping and dependency among contributions.

non-dependent) and *(non-overlapping, non-dependent)* configuration for (c) and (d) respectively.

Given the above, we need a way to acknowledge the direct and indirect relationships that exist between contributions, and from that generate reliable grouping method.

Similarity Metric: We propose a similarity metric for comparing contributions that takes into account the overlapping between modified elements, but also the relationship between elements that comes from their time dependency. Given two contributions c_i and c_j with associated set of modified files M_{c_i} and M_{c_j} respectively. Then, we propose the following similarity metric: $S(c_i, c_j) = DS(c_i, c_j) + IS(c_i, c_j)$, where $DS(c_i, c_j)$ represents the direct similarity that comes from the overlapping subset of elements modified by both contributions and is defined by: $DS(c_i, c_j) = \frac{M_{c_i} \cap M_{c_j}}{max\{\|M_{c_i}\|, \|M_{c_j}\|\}}$ and $IS(c_i, c_j)$ represents the indirect similarity between contribution that derives from the existent dependency between artifacts. In this case, given an element $a \in M_{c_i}$ and $b \in M_{c_j}$, such as $a, b \notin M_{c_i} \cap M_{c_j}$, if there is a path from a to b in the dependency structure of the software system under study, then the similarity is given by $\sum_{a \in (M_{c_i} \setminus M_{c_j})} \sum_{b \in (M_{c_j} \setminus M_{c_i})} \frac{1}{\|path_{a \to b}\|}$. For this case, it is assumed $\|path_{a \to b}\| \neq 0$ and that if exists, there is one unique path from a to b. If there are more than one, the shortest path is selected.

One of the requirements for the grouping algorithm is that the resulting subsets of contributions should follow a defined order. This order is the key element that supports the idea is *cascading*: if contribution A *depends on* or was *influenced by* contribution B, then it is necessary that A was generated before B. The time interval between contributions is assumed to represent the period on which the community noticed the first contribution and based on their understanding and usage, decided to conduct new modifications on the code, which will conform the second contribution.

Therefore, the question is how to incorporate this dimension into the clustering methodology in order to generate feasible cascades? As a first step, it will necessary to order the contributions before performing the any grouping activity.

As literature does not provide an definitive way to achieve this goal [7], we propose an exploratory analysis consisting of the comparison between two methods with different nature. We expect to obtain insight about the performance of the methodologies in relation with the specific characteristics of the data.

K-means with post-ordering: This method consists of two phases. Firstly, the contributions are grouped through standard K-means using the similarity metric presented above. The number of clusters is chosen by minimizing the ratio between intra and extra cluster distance. Then, the second phase consists of taking each resulting cluster and order the contributions in relation to the time they were submitted. This part involves a specific filtering, as a threshold must be specified.

Heuristic approach: This method is based on a set of ad-hoc rules. It initially takes the set of pull requests available from the repository, ensuring it is ordered.

The algorithm uses three parameters to generate the clusters: Firstly, the time interval between contributions Δ_t, assuming that although two contributions could be similar in terms of the files they modify, if they are not close in terms of time, the should not be considered as part of the same cascade. The second parameter is the accumulated time of the cascade, which must be less than a fixed T_c. This assumption is based on the observation that real world cascades tend to fade over time and have a defined ending. The third parameter is the minimum level of similarity S_{min} the that a contribution needs to have with the previous added one in order to be incorporated to the cascade.

The algorithm begins by taking the first contribution and comparing it with the second. If the three constrains are satisfied, both contributions are linked. Then, the process continues analyzing the second and the third contribution, and so on. Then, the algorithm takes the second contribution as the initial one and the process is conducted again. This leads to several candidate configurations. The criteria used is to choose the configuration that minimizes the ratio between the number of cascades and their average length.

2.3 Cascade Feature Selection

Project Features: The project is the ecosystem in which all the interactions and activities are held. Therefore it is key element to consider in our study. In that sense, we are considering all the elements the influence the generation of new activities. Features in this category include number of watchers, commits and average level of activity in the pull requests discussions.

Contribution Features: Contributions, in the form of *pull requests* are the basic unit of activity considered in this study. They provide the details of how the developers modify the source code of the project, such as the files added or changed. Additionally, it provides all the interaction generated while the contribution was reviewed by the core team. We make the distinction between the initial contribution and the set of subsequent contributions. Features in this category include timestamps of creation, merge and closing, number of commits included, the amount of activity (in terms of people, review actions and comments) and the proportion between files added, modified or deleted, among others.

Contributor Features: The contributor is the developer that visualized an opportunity of improvement and decided to submit a new feature or bug fix to the We also make the distinction between the contributor the responsible for the initial contribution and the ones that are part of the subsequent activities. Features in this category include level of ownership, number of projects the contributor is working on, social features (watchers, favorites), among others.

2.4 Prediction Problem Definition

In order to avoid the issues that previous works on cascade prediction presented (unbalanced classes and over-representation of large instances), we modeled the problem as a binary classification. Given a set of linked activities, with length of A elements , we observed the first one, namely the igniter, and the $k-1$ following, comprising a subset of k elements, with $k < A$. Then, our task is to predict if the size if the length of the set of linked activities will reach $f(k)$, which is the median size of all the set of linked activities that reach *at least* k activities. This classifications task is carried out using logistic regression.

3. Exploratory Study

We collected data from Github social coding platform, specifically using the *pullreqs* project, which comprises developer contribution from several important projects on Github (See [6] for more details). For this exploratory analysis, we chose six relevant projects and from their historical data the analysis was performed.

The procedure consisted on choose an starting size and sequentially increment it while analyzing the overall accuracy. Given the data obtained, we began with $k = 3$ and explore until $k = 14$.

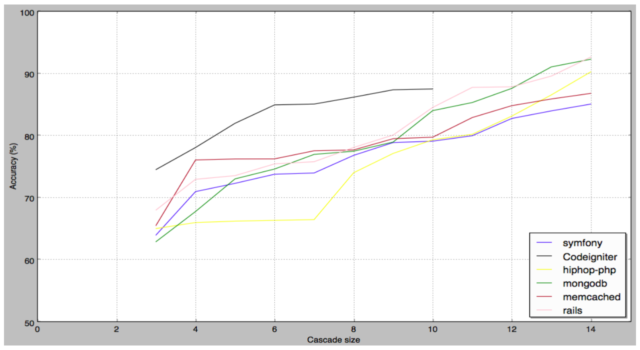

Figure 3: Exploratory results based on Github activity.

As seen from the Figure 3, the accuracy increases as the cascade get longer, but in most cases the rate of increase is reduced, this means that, although the information gathered over time is useful to improve the predictability of the cascade, at the same time, the current model is not capable of fully handle the variability. In terms of how the contribution dynamics influence the performance of the approach, we found initial evidence that, given a initial contribution, if the subsequent activity begin in a short time, the cascade tends to be larger. We hypothesize that an intense stream of contributions boosts the overall activity and collaboration among the team.

In terms of the categories of features that have more explanatory power, the set of *contribution* features outperforms the rest of the sets (average of .78), followed by *contributor* features (average of 0.66).

Although this is a limited study, it can be seen that the prediction of the dependency should be understood as a continuous process, as simply considering the initial stage lead to poor performance. This setting is useful in the context of OSS, as core members can monitor activity and estimate accurately level upcoming level of contributions.

References

[1] F. Beck and S. Diehl. On the congruence of modularity and code coupling. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 354–364, New York, NY, USA, 2011. ACM.

[2] D. Beyer and A. Noack. Clustering software artifacts based on frequent common changes. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, pages 259–268, May 2005.

[3] J. Cheng, L. Adamic, P. A. Dow, J. M. Kleinberg, and J. Leskovec. Can cascades be predicted? In *Proceedings of the 23rd international conference on World wide web*, pages 925–936. International World Wide Web Conferences Steering Committee, 2014.

[4] J. Choi, J. Choi, J. Y. Moon, J. Hahn, and J. Kim. Herding in open source software development: an exploratory study. In *Proceedings of the 2013 conference on Computer supported cooperative work companion*, pages 129–134. ACM, 2013.

[5] G. Gousios, M. Pinzger, and A. van Deursen. An exploration of the pull-based software development model. In *ICSE '14: Proceedings of the 36th International Conference on Software Engineering*, jun 2014. To appear.

[6] G. Gousios and A. Zaidman. A dataset for pull request research. In *MSR '14: Proceedings of the 11th Working Conference on Mining Software Repositories*, may 2014. To appear.

[7] E. Keogh and J. Lin. Clustering of time-series subsequences is meaningless: implications for previous and future research. *Knowledge and information systems*, 8(2):154–177, 2005.

[8] J. Lerner and J. Triole. The simple economics of open source. Working Paper 7600, National Bureau of Economic Research, March 2000.

[9] A. Meneely and L. Williams. Secure open source collaboration: an empirical study of linus' law. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 453–462, New York, NY, USA, 2009. ACM.

[10] T. Zimmermann and C. Bird. Collaborative Software Development in Ten Years: Diversity, Tools, and Remix Culture. In *Proceedings of the Workshop on The Future of Collaborative Software Development*, 2012.

[11] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *Software Engineering, IEEE Transactions on*, 31(6):429–445, June 2005.