Predicting Fork Visibility Performance on Programming Language Interoperability in Open Source Projects

Bee Bee Chua
University of Technology, Sydney
Australia

***Abstract*—Despite a variety of programming languages adopted in open source (OS) projects, fork variation on some languages has been minimal and slow to be adopted, and there is little research as to why this is so. We therefore employed a K-nearest neighbours (KNN) technique to predict the fork visibility performance of a productive language from a pool of programming languages adopted in projects. In total, 38 showcase OS projects from 2012 to 2016 were downloaded from the GitHub website and categorized into different levels of programming language adoption clusters. Among 33 languages, JavaScript is one of the popular languages that adopted by community. It has been predicted the language chosen when fork visibility is high can increase project longevity as a highly visible language is likely to occur more often in projects with a significant number of interoperable programming languages and high language fork count. Conversely, a low fork language reduces longevity in projects with an insignificant number of interoperable programming languages and low fork count. Our results reveal the survival of a productive language is in response to high language visibility (large fork number) and high interoperability of multiple programming languages.**

***Keywords*—Open Source Programming Languages; K-nearest neighbors (KNN) Algorithm; interoperability; survivability**

I. Introduction

Programming languages constantly evolve to meet the demand of the software development industry. However variation of programming languages adopted in open source (OS) projects must comply with other programming languages so that developers can fork (copy) language files into their own local development environment. To ensure interoperability, programming languages must be expressive, generic and compliant, otherwise developers will not be interested in downloading or forking new OS libraries, as the frameworks are not compatible with their environment. There are different ways to define programming language success, with programing language interoperability performance being a major contributor to success. Despite this, unfortunately, most languages are not interoperable.

To understand when and why developers would fork a programming language file, language needs and motivation are two important factors. Some developers may fork a language because it is a new language that compiles with the original language, while other developers may fork a language because it is a subset of the original language, with features added, removed or amended.

In spite of these motivating reasons to inspire developers to fork languages, many programming languages are experiencing a ‘fork crisis’, that is, they have low or minimal fork counts. This may be due to social factors [1]-[3] and environmental reasons [4]-[6], or the languages may lack expressiveness, be too generic or have compliance with the original or other languages. Interestingly, many OS project owners tried to increase programming language interoperability by adopting different programming languages; however this does not seem to increase forking.

Our motivation for this paper is firstly to make an intelligent recommendation system for developers and project owners to adopt programming languages that are compliant with other language interoperability. Secondly, to understand how a productive language fork may be affected by low programming language interoperability and low compliance with many programming languages’ interoperability.

This paper is organised into the following sections: Section 2, literature around language forking prediction, the problem and research questions; Section 3 research methodology on KNN algorithm, data quantisation methods and a case study of OS projects; Section 4 results, Section 5 outcomes of the four scenarios tested; and lastly, justification and conclusions.

II. Programming Language Forking

A. Language Forking Prediction Problem and Research Questions

We investigated whether it was possible to predict with reasonable accuracy the fork visibility performance of any programming language with respect to interoperability compliance. In addition, we sought to determine the probability of new projects adopting a productive language where fork visibility performance is impacted by low versus high programming language interoperability.

Two research questions were developed to address these aims:

1) How can we predict, with reasonable accuracy, a programming language fork visibility performance in projects that is in compliance to other languages interoperability?

2) For a new project, how can we predict productive programming language fork visibility performance based on the level of programming language interoperability?

In this paper, we define a ‘more’ interoperable programming language project as a language that has more healthy forks in the majority of programming languages, and a ‘less’ interoperable programming language project as one with fewer healthy forks in each language.

III. Methods and Dataset Preparation

A. *K Nearest Neighbour (KNN) Algorithm*

The KNN algorithm is based on representation of statistics and distributions in training data. While the method was first discovered in 1961 by a group of American researchers who showed it works effectively on actual instances of training data [7], it remains unpublished. It has since been applied to machine learning and data mining, and more recently has successfully been applied in education research to predict student learning success and failure rates [8]-[12]. The KNN method is effective at predicting different types of data, is simple and versatile, and handles noisy or incomplete data, when in many situations a classification is required [13]-[17].

The baseline KNN predicts the fork performance of a given project by first calculating the actual project (project being predicted) similarity to all instances in the training set and finds the K most similar ones. The similarity is calculated with a simple Euclidean distance between the features of the test subject and corresponding features of each instance in the training set [12].

In this study, KNN was used to predict fork visibility performance of languages that were adopted as interoperable language in projects to differing degrees (‘more’ or ‘less’). Firstly the algorithm applied Euclidean distance formula (see Fig. 1) to calculate the distance of a productive language fork for less adopted interoperable language projects. X refers to the number of language repositories created in the project and Y refers to the number of programming languages adopted in the project. X1 is the actual number of language repositories from 38 project showcases and X2 is the predicted number of new project language repositories. Similarly, Y1 and Y2 are the actual and predicted numbers of programming language from the 38 project showcases and the new project.

We classified the outcome of that algorithm into two categories: 1) JavaScript in a project with low fork visibility; and 2) JavaScript in a project with high fork visibility. Next, we used K=3 to predict the language project on JavaScript fork visibility outcomes.

$$\mathrm{D}(x,y)=\sqrt{(x1-y1)2+(x2-y2)2}$$

Fig. 1. KNN equation.

B. *Case Study: Showcase Projects*

Of the 40 OS show case projects available on GitHub on from January 2012 through August 2016 (www.github.com), 38 projects have complete information such as the type of programming languages and the fork count. We rejected 2 projects because of some programming languages were not stated (unknown). As our goal was to predict the language fork visibility performance, defined as success or survival of different programming languages in a project, the 38 projects were classified into types of projects and by different levels of programming languages (Fig. 2).

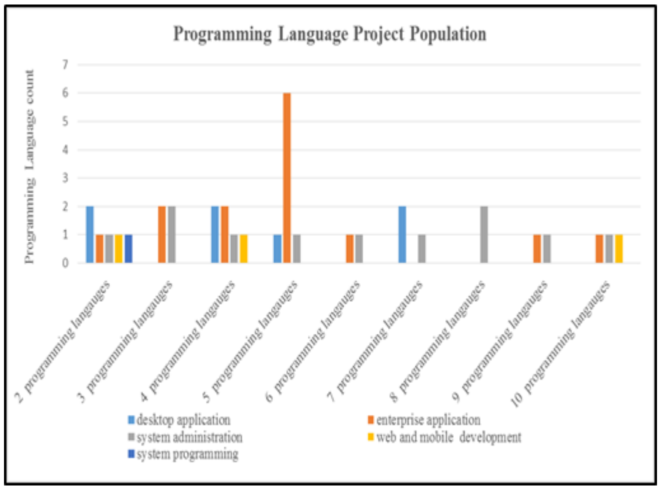

Fig. 2. Programming language project population.

The types of projects ranged from desktop application, enterprise application, systems administration, systems programming and website development.

Next, we categorized productive programming languages by types of programming language tier level according to the TIOBE programming community index, which ranks various programming languages [18], [19]. Fig. 3 shows that projects adopted from 2 to 9 programming languages, and JavaScript was the most popular.

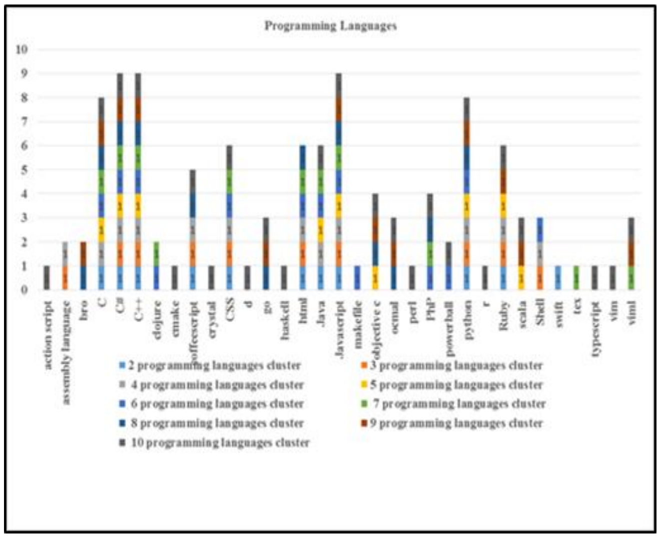

Fig. 3. Tier levels of programming languages.

C. *Programming Language Fork Visibility Performance and Data Quantisation*

Prior to applying the KNN algorithm (see below), we first identified the features of programming language fork visibility performance that responded to programming language interoperability. These included individual programming language type, the number of individual programming languages adopted per project, the individual language repository number, and individual language fork frequency, when available (published on the project webpage). Then, due to the large quantity of fork counts, the data underwent quantisation, with each feature weighted as per Table 1.

TABLE I. PROGRAMMING LANGUAGE FORK PERFORMANCE FEATURE

Feature	Range	Weight	
		Min	Max
Number of adopted programming languages	1–10	0.1:1	1.0:10
Adopted language repository file number	1–10	01:1	1.0:10
Specific language fork number	300–200,000	0.01:1–500	0.2:200,000

Quantisation produced a total number of 2652 data features. An example of each project data that converted to data quantisation as follows to each field as: number of adopted programming langauges, adopted language repository file number, from specific programming language fork number 1 to number 33.

0.1,0.2,0.01,0.1,0.01,0.3, 0.01,0.0001,0,0,0,0,0,0,0.0.1, 0.0001,0,0, 0,0,0.0001,0,0,0.01,0.0001,0.01,0.1,0,0,0,0.1,0

D. *Averaging Programming Language Number and Programming Language Fork Count*

To confirm the programming language fork visibility performance, we set a threshold on programming language number and fork count size, with minimum and maximum values. To support the threshold, we derived an equation to determine the threshold outcome based on two further equations: 1) Average Programming Language Number (APLN); and 2) Average Programming Language Inter-operability (APLI), for the APLN and APLI, the formulas were:

$$APLN \quad \frac{\text{Total number of project language number}}{\text{Total number of project in the case study}} \qquad (1)$$

$$APLI \quad \frac{\text{Total number of project language number}}{\text{Total number of project in the case study}} \qquad (2)$$

Next, we compared each APLN against the APLI in the project. If the APLN score was greater than the APLI score then the project was defined as having adopted high programing language interoperability. Conversely, if the APLN was less than the APLI score then the project was defined as having adopted low programming language interoperability.

IV. RESULTS

Fig. 4 shows a simple example illustrating KNN with two features (programming language fork size count as the x axis and programming language number as y axis) to find the JavaScript visibility performance.

The justification on JavaScript as it produces many libraries and frameworks on OS projects that are compliant for cross-platform integration. Moreover, the JavaScript language community is large because it is familiar to developers who learned it during training and qualification. In the context of this paper, we were interested to find out the predicted outcomes for JavaScript fork performance on low and high programming language interoperability for a new project.

We generated four scenarios to predict their outcomes using the KNN algorithm. The first scenario was a project that was likely to receive low fork count in JavaScript, which adopts low average programming language interoperability (APLI). The second scenario was a project likely to experience high JavaScript fork in the adopted low APLI. The third scenario was a project with low JavaScript fork in a high APLI, and the fourth scenario was high JavaScript fork in a high APLI.

Scenario 1: JavaScript low fork visibility performance with low adopted programming language interoperability

The first scenario was a new project that adopted very low programming language interoperability, including JavaScript. Fig. 4 shows the new project (orange circle) distance is close to projects A, C and G. By majority voting, project C was predicted as the nearest to the new project, that is, the new project JavaScript language fork was predicted to be low if the adoption of programming languages interoperability was low.

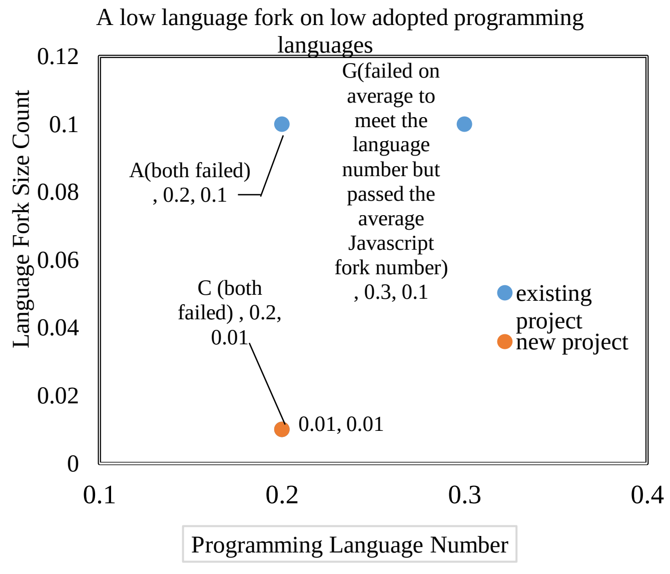

Fig. 4. Scenario 1: JavaScript low fork visibility performance in a low APLI.

Project A was a website development that had adopted JavaScript and Ruby and, based on their fork population; it was very close to the JavaScript fork size on the new project. Project C, on the other hand, was an enterprise application and adopted only 2 programming languages – JavaScript and CSS. The project failed to receive high fork attention because CSS is used for formatting structured content on HTML documents. As a result, it is less interesting to developers as a problem-solving technique. For Project G, despite having JavaScript, Python and HTML as marked up languages adopted, they face survival problems being unable to find developers to fork the language file, possibly because Python is less compliant with JavaScript [6], thus lessening JavaScript forking.

Overall, these project languages failed to pass the average adopted programming language interoperability levels and average JavaScript fork count size. By majority voting – where K=3 – a new project was predicted to fail in a low adopted programming languages and low JavaScript fork environment.

Scenario 2: JavaScript high fork visibility performance with low adopted programming language interoperability

The second scenario outlined JavaScript high fork visibility performance in a low APLI, which was the reverse of the first scenario. Fig. 5 shows the new project (orange circle) is close to projects A1, Q and D1. We applied K=3 which resulted in a tied vote, with a different outcome on the three projects. Project A1 had a sufficient APLI number but failed to generate a high JavaScript fork. Project Q failed on the APLI but passed on the average number of JavaScript forks. In contrast, Project D1 satisfied both conditions, passed APLI and average number of JavaScript forks. However as the data set was small no one single outcome can predict whether a new project would be likely to be near to an existing project. We further examined each project cause, finding that JavaScript language files added new features that attracted developer attention.

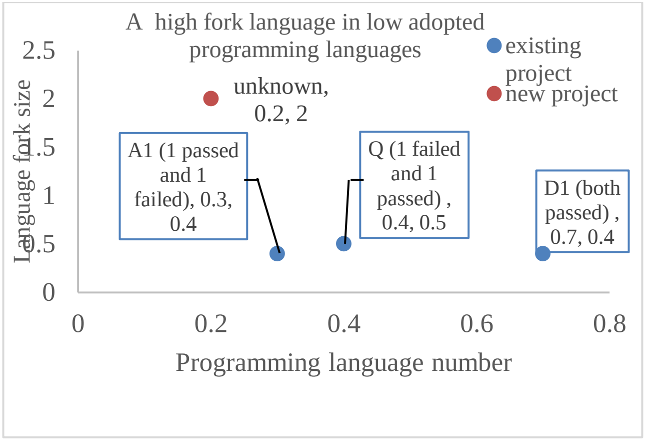

Fig. 5. Scenario 2: JavaScript high fork visibility performance in a low APLI.

Scenario 3: JavaScript low fork visibility performance with highly adopted programming language interoperability

The third scenario was a new project with high APLI and low fork count on JavaScript. Based on the majority voting, the three projects predicted to the nearest distance of the new project were J1, L1 and K1 (Fig. 6). Successfully all passed both the average programming language interoperability number and the average JavaScript fork number. The results showed that low language fork can arise in a project with some languages adopted with weak compliance to JavaScript. In Scenario 3, non-JavaScript language files focused on back-end development; as such they were of core project value. Consequently, it has a high impact on JavaScript developers' fork behaviour to download and fork less the JavaScript files.

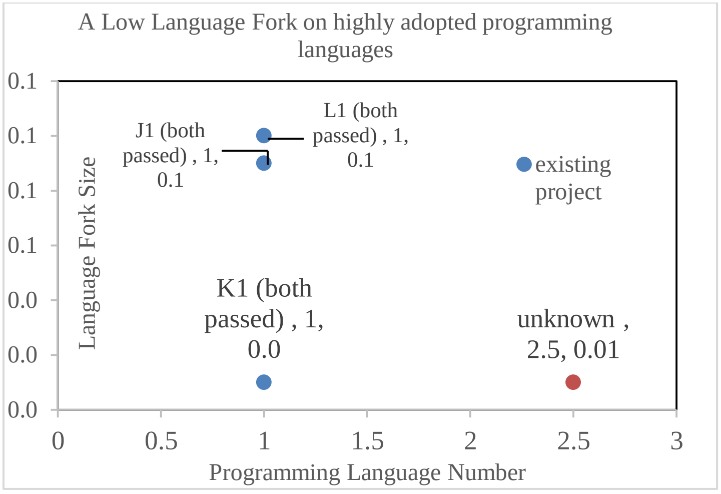

Fig. 6. Scenario 3: JavaScript low fork visibility performance in a high APLI.

Scenario 4: JavaScript high fork visibility with highly adopted programming language interoperability

The fourth scenario was a new project that adopted a variety of programming languages; the JavaScript language is one of the most well-known languages that contain a high fork count. Fig. 7 shows the distance of a new project status (orange circle) and existing projects D1, Q and L1. The three existing projects passed the average adopted programming language interoperability number and the average JavaScript fork count. We applied K=3 to detect the possible outcome for the new project. The result shows by majority voting in this case all 3 projects have the same outcome and they are predicted the nearest projects to the new project.

These projects seemed to perform better because they were compliant with other programming languages, such as Ruby, PhP, Python C and C++. As JavaScript shows a high connectivity with Ruby and PHP [20], JavaScript can fetch a high fork count from developers. From the project development perspective, the topic domain or field interest to developers, and the selective programming languages, contribute to the high fork frequency. From our observation on the three projects' fork aggressiveness, the languages adopted in these projects are compatible to cross platforms.

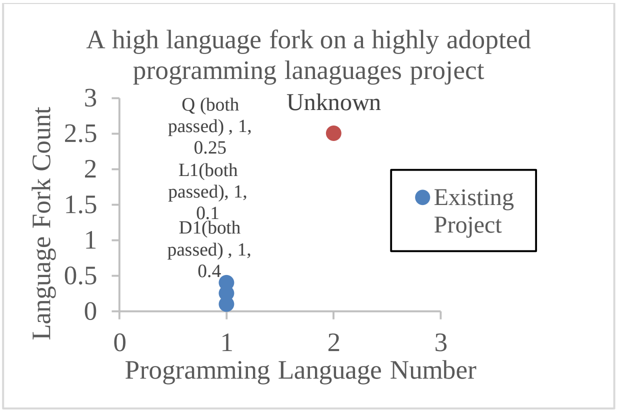

Fig. 7. Scenario 4: JavaScript high fork visibility performance in a high APLI.

V. Justification

1) Positioning a productive language in a pool of compliant language interoperability:

Our previous work [21] introduced a technique to detect the chance of programming languages used in Apache, Mozilla and Ubuntu surviving from a forking perspective. The current work from the evidence, the productive language, JavaScript, showed less difficulty to survive when placed in a pool with low APLI. In addition, a low survival of JavaScript could be expected in conjunction with high APLI because JavaScript is less compliant with other languages' interoperability, except Ruby and PhP [19], [20].

2) Programming languages fork visibility performance: A new perspective on survivability and longevity:

The scenario-based evidence presented here provides a new perspective on developers' fork behaviour – particularly on programming language interoperability number adopted in a project and how they might influence each other, especially the productive languages. However, there is no certainty on which programming language can survive longer in terms of emerging technology, except it must be compliant with other language interoperability. Due to an increased change in emerging technology – such as mobile application and cloud development – more projects will increasingly add more programming languages for interoperability. As such, the more a language is compliant, the more likely a language will increase fork visibility, and in turn increase language survivability.

Previous work [22]-[29] showed OS variables' impact on developers' fork behaviour is generally related to project topics and domains, developers' language preference, and programming language popularity. However, our findings show another possible cause, that is, poor fork visibility in a low APLI is less compliant. A language with low fork visibility is likely to decline its longevity and survivability whereas a language with high fork visibility is likely to increase its longevity and survivability, which will serve to keep the project viable and accessible by developers. Understanding programming language fork success or failure draws a new perspective out of the literature, highlighting that fork success is highly dependent on specific language domination [1] and/or a productive language [20].

3) The relationship between visibility and vulnerability in the context of open source programming languages:

The term 'visibility' is described in the context of meteorology as transparency of air, in the dark, etc. In a disruptive or sustainable technology, visibility is a metric used to determine factors such as project vulnerabilities, consumer confidence, or purchasing pattern or behaviour. In the context of OS programming languages, visibility exposes the vulnerability of a language, which can become less significant as a result to sustain frameworks and libraries, front-end, back-end, etc. As such, new programming languages with better implementation performance are likely to dominate and replace existing language source codes.

VI. Conclusion

This research focused on applying an algorithm to a case study of four scenarios. The preliminary findings require further validation in a larger dataset to examine programming language strength, in terms of compliance, compatibility and connectivity. This paper introduced a new perspective to OS programming language survivability research, particularly the fork visible performance that different programming languages exhibit and their interoperability performance across different ecosystems and environments.

References

[1] Nyman, L. 2013. Freedom and forking in open source software. Proceedings of the Nordic Academy of Management Conference ,Reykjavik, Iceland

[2] Tsay, J. Dabbish, L. and Herbsleb, J. 2014. Influence of Social and Technical Factors for Evaluating Contribution in Github, Proceeding of ICSE'14, Hyderabad, India, ACM.

[3] Khondhu, J. Capiluppi.A , Stol, K.J. 2013. Is It All Lost? A Study of Inactive Open Source Projects. In Proceedings of the 9th International Conference on Open Source Systems

[4] Crowston, K. Howison, J. and Annabi, H. 2006. Information Systems Sucess in Free and Open Source Development : Theory and Measures " Software Process and Practice, Vol. 11, No 2, Pp. 123-148

[5] V. Midha and P. Palvia, "Factors affecting the success of open source software," The Journals of Systems and Software, vol. 85, no. 4, pp. 895–905, 2012.

[6] S. Comino and F. M. Manenti, "Government policies supporting open source software for the mass market," Journal Review of Industrial Organization, vol. 26, no 2, pp. 217–240, 2005

[7] Johns, M. V. (1961) An empirical Bayes approach to non-parametric two-way classification. In Solomon, H., editor, Studies in item analysis and prediction. Palo Alto, CA: Stanford University Press

[8] Tanner, T. and Toivonen, H. (2010). Predicting and preventing student failure – using the k-nearest neighbour method to predict student performance in an online course environment. International Journal of Learning Technology 5(4):56–377. https://www.cs.helsinki.fi/u/htoivone/pubs/ijlt2010.pdf

[9] Ishii, N., Hoki, Y., Okada, Y. and Bao, Y. (2009). Nearest neighbor classification by relearning. In: Proceedings of the 10 International Conference on Intelligent Data Engineering and Automated Learning (IDEAL'09**)**, pp. 42–49.

[10] Kotsiantis, S., Pierrakeas, C. and Pintelas, P. (2003). Preventing student dropout in distance learning systems using machine learning techniques. In: Proceedings of 7th International Conference on Knowledge-Based Intelligent Information & Engineering Systems, Lecture Notes in Artificial Intelligence, Springer-Verlag. 2774:267–274.

[11] Minaei-Bidgoli, B., Kashy, D.A., Kortmeyer, G. and Punch, W.F. (2003). Predicting student performance: an application of data mining methods with an educational Web-based system. In: Proceedings of the 33rd Annual Frontiers in Education,1:T2A–18.

[12] Shih, B. and Lee, W. (2001). The application of nearest neighbour algorithm on creating an adaptive on-line learning system. In: 31st Annual Frontiers in Education Conference, 1:T3F–10–13.

[13] Manning, C., Raghavan, P. and Schütze, H. (2008). Introduction to Information Retrieval. Cambridge University Press: Cambridge, UK.

[14] Dumais, S., Platt, J., Heckerman, D. and Sahami, M. (1998). Inductive learning algorithms and representations for text categorization. In: Proceedings of the International Conference on Information and Knowledge Management,. pp. 148–155.

[15] Dumais, S., Platt, J., Heckerman, D. and Sahami, M. (1998). Inductive learning algorithms and representations for text categorization. In: Proceedings of the International Conference on Information and Knowledge Management, pp. 148–155.

[16] Han, Y. and Lam, W. (2006). Exploring query matrix for support pattern based classification learning. Advances in Machine Learning and Cybernetics, Lecture Notes in Computer Science 3930:209–218.

[17] Zou, Y., An, A. and Huang, X. (2005). Evaluation and automatic selection of methods for handling missing data. In: Proceedings of the IEEE International Conference on Granular Computing, 2:728–733.

[18] Cover, T. and Hart, P. (1967). Nearest neighbour pattern classification", *IEEE Transactions on Information Theory* 13(1):21–27.

[19] TIOBE. TIOBE programming community index definition. 2016, http://www.tiobe.com/index.php/content/ paperinfo/ tpci/tpcidefinition.html

[20] Bissyande, T.F., Thung, F., Lo, D., Jiang, L.X. and Réveillère, L. (2013). Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In: Proceedings of COMPSAC '13: 2013 IEEE 37th Annual Computer Software and Applications Conference, 22–26 July, 2013, Kyoto, Japan. pp. 303–312. Research Collection School of Information Systems.

[21] Chua, B. 2015, 'Detecting Sustainable Programming Languages through Forking on Open Source Projects for Survivability', IEEE, The 26th IEEE International Symposium on Software Reliability Engineering (ISSRE) 2015 in conjunction with a WOSAR workshop, IEEE, Gaithersburg, USA, pp. 120-124.

[22] Samoladas, I. Angelis, L. and Stamelos, I. 2010. Survival duration on the duration of open source projects. Journal of Software and Information Technology, Vol.52, No.1, Pp 902-922.

[23] Chen, S. 2010. Determinants of Survival of Open Source Software: An Empirical Study. Academy of Information and Management Sciences Journal, Vol.13, No.2, Pp119-128.

[24] Wang, J.2012.Survival factors for Free Open Source Software projects: A multi-stage perspective," European Management Journal, Vol.30, No.1, Pp352-371.

[25] Wu, J. and Tang, Q. 2007. Analysis of Survival of Open Source Projects: a Social Network Perspective. In proceedings of Australian Conference of Information Systems (ACIS).

[26] Angelis, L. Sentas, P. 2005. Duration Analysis of Software Projects. In Proceedings of the 10th Panhellenic Conference on Informatics, 258-269.

[27] Raja, U. and Tretter, M. J. 2012. Defining and Evaluating a Measure of Open Source Project Survivability. Journal of IEEE ,Transactions on Software Engineering, Vol.38, No.1,Pp163-174.

[28] Chengalur-Smith, I., Sidorova, A. and Daniel, S. Sustainability of Free/Libre Open Source Projects: ALongitudinal Study. Journal of Association For Information Systems (JAIS), Vol.11, No. 11/12.Pp657-683.

[29] Oskar, J., Gruszka, B., Jaroszewicz, S., Bukowski, L. and wierzbicki, A. 2014. GitHub Projects. Quality Analysis of Open-Source Software. In the proceeding of 6h International Conference, Soclnfo. Barcelona, Spain,Lecture Notes in Computer Science Volume 8851.