Predicting Open Source Programming Language Repository File Survivability From Forking Data

Bee Bee Chua, Ying Zhang
University of Technology, Sydney, Centre for Artificial Intelligence
Sydney, Australia

ABSTRACT
Very few studies have looked at repositories' programming language survivability in response to forking conditions. A high number of repository programming languages does not alone ensure good forking performance. To address this issue and assist project owners in adopting the right programming language, it is necessary to predict programming language survivability from forking in repositories. This paper therefore addresses two related questions: are there statistically meaningful patterns within repository data and, if so, can these patterns be used to predict programming language survival? To answer these questions we analysed 47,000 forking instances in 1000 GitHub projects. We used Euclidean distance applied in the K-Nearest Neighbour algorithm to predict the distance between repository file longevity and forking conditions. We found three pattern types ('once-only', intermittent or steady) and propose reasons for short-lived programming languages.

Author Keywords
Programming language, survivability, forking, open source, K-Nearest Neighbour, Euclidean distance, prediction

ACM Classification Keywords
Programming language, survivability, forking, open source, K-Nearest Neighbour, Euclidean distance, prediction

1. INTRODUCTION

Programming language survivability can be predicted in different ways, with different evaluative methods generating different predictive results. Forking is sometimes ignored when predicting repositories' programming language survivability in GitHub, as the GitHub forking function is an essential mechanism to assist open source (OS) developers to quickly code software with support of the internal and external community.

OpenSym '19, August 20–22, 2019, Skövde, Sweden

ACM ISBN 978-1-4503-6319-8/19/08…$15.00
https://doi.org/10.1145/3306446.3340827

Whether a programming language can survive (perform) or not is highly dependent on forking performance. Each repository file is tied to a programming language that provides developers the freedom to copy and fork the file. Forking features include speed, size and type. Speed refers to the forking period in days, weeks or months; size refers to the number of developers who fork the file; and type refers to source code file characteristics, such as programming language, license compliance, etc

However forking has some challenges; for example, forking performance could be a "high demand but low supply", "low demand and supply" or "low demand but high supply" situation. For example, "high demand but low supply" may reflect using a popular programming language but the repository is not forked by many developers. Conversely, "low demand and supply" may be a niche programming language, developers and market, e.g., using R language for statistics and data analytics. "Low demand but high supply" may be a new and popular programming language – Swift, Objective C – that developers would be more likely to adopt because of language migration.

It is unclear what causes uncertainty in low or high forking count, affecting programming language survivability. Our research therefore focuses on programming language survivability. This research is critical as an increasing number of repository files are adopted to sustain programming languages, but creators may not be able to find the right developers to fork their language. Further, there is a recent decline in forking as correct programming languages are not being adopted onto repository files. To tackle these issues, our goal in this paper is to report evidence of the effect of forking in programming language repository files.

We are the first researchers to analyse a large forking dataset for developer forking behaviour based on repository file characteristics to predict sustainable programming language survivability. We are also the first to adopt a machine learning method, K-Nearest Neighbour (KNN), to predict sustainable programming language survivability and introduce a robust method to evaluate OS file forking success ability.

Here Section 1 provides the research overview; Sections 2 and 3 describe forking, patterns and programming language importance; Section 4 explains the KNN and Euclidean Distance methods and the GitHub dataset; and Section 5

outlines the programming language repository file categorisation and fork pattern classifiers. Results are presented in Sections 6, and 7 , with discussion, conclusions and future work presented in Section 8.

2. CONCEPT OF FORKING

Forking can be defined in different ways. Nyman and Mikkonen [1–3] defined it in a project context, where developers copy source code from one software package to develop an independent project. Ikuine and Fujita [4] defined project forking as the continuous development of a software. Fung, Aurum and Tang [5] defined social forking to identify relationships within communities, and studied how forks are used to facilitate OS software development. In our paper, we define language forking as a repository language that is copied by other developers.

There are also different ways to define programming language success, with programming language inter-operability performance being a major contributor to success. Despite this, most languages are not interoperable [6–8]. Language needs and developer motivation are two important factors when investigating when and why developers may fork a programming language file. For example, some developers may fork a language because it is a new language that complies with an original language, while other developers may fork a language because it is a subset of the original language, with features added, removed or amended. As such, we need to understand developer forking motivation. There are currently a broad range of perspectives on OS developer motivation, ranging from individual to communities, and fork consequences on projects and organisations.

Although a variety of research methods have been adopted to predict OS software popularity, sustainability and survivability [1–6,9–16], these methods are less useful for predicting programming language forking survivability. These research methods include surveys, interviews, content analysis and empirical studies that are subjective and potentially biased. For example, data samples were not large, reducing accuracy; data analyses and interpretation could be subjective or biased; and the study designs were unable to handle large data sets, unlike machine learning techniques that work effectively with an abundance of data to leverage for training and testing.

2.1 Understanding Open Source Forking

To eliminate the disparity view of forking, a group of researchers [12,17] systematically reviewed the literature [18,19] and performed a content analysis [20,21] to analyse OS developers' forking motivation, interpretation, category and consequence. They found forking can be categorised into seven types: OS; project; software; social; code; programming language; and file repository file forking. The categories are outlined below and are quite similar apart from different fork behaviours.

2.1.1 Open source forking

The early 1990s saw a plethora of research on OS developer motivation. Krogh and colleagues [15] reviewed seven years' of publications across 40 researchers to identify reasons that motivated OS developers to voluntarily spend time fixing and contributing source code. They classified these into intrinsic, internalised intrinsic and extrinsic motivation. Intrinsic motivation included ideology, altruism, kinship and fun; internalised intrinsic included reputation, reciprocity, learning and own-use; and extrinsic included being paid or career building.

2.1.2 Project forking

Nyman and Mikkonen [1–3] proposed that a project fork occurs when software developers copy source code from one software package and use it to develop an independent project. Forking therefore produces an independent version of the system that is maintained separately from its origin. They quantified project forking as the number of original projects forked by developers, comparing the number of original projects versus forked projects in GitHub.

They looked at forking behaviour in the context of forked project survivability. Researchers are still seeking to further understand how forking impacts the original forked project and Nyman and Mikkonen provide real-life examples of current high profile OS projects that either started from a fork or are common targets for forking [1–3].

2.1.3 Software forking

Ikuine and Fujita defined forking as the continuous development of software [4]; that is, the software continues to be developed by the original developer or other developers. When other developers copy the file, the original developer must share the source code. Software forking focuses on the product itself, such as Microsoft or Facebook software, and email applications.

2.1.4 Social forking

In their study of nine JavaScript development communities in GitHub, Fung, Aurum and Tang defined social forking as the highest amount of forks required to identify relationships within them, and studied how forks are used to facilitate OS software development [5]. They analysed approximately 8,000 forks from almost 7,000 developers across different communities, with the most active developers making contributions to multiple communities.

Their research indicated that forks are actively used by the development community to fix defects and experiment with new features. What separates forks from normal branching is that changes do not need to be promoted in the original upstream project and can live in a separate fork that can still change and improve, independent of the original project. Further, a branch is that a fork can originate from either a subset of the forked predecessor's artefacts or from multiple predecessors' artefacts. A branch in turn is a copy of all the predecessor's artefacts [5].

2.1.5 Code forking

Code forking is defined as a forked project copied from the existing code base and moved in a direction different from the project leadership. Forking the code base allows developers to leverage existing functionality while also addressing new requirements. Although flexible, code forking has inherent difficulties, such as maintenance, evolution, and social factors within the development community. A broad definition of a code fork is when the code from an existing program serves as a fork it is the basis for a new version of the program [4,14,22]; more specifically, a version that seeks to continue to exist apart from the original.

2.1.6 Programming language forking

Chua [11–12,17] examined language forking from the perspective of programming language adoption in projects by project owners. She found three projects where Apache, Mozilla and Ubuntu JavaScript languages were actively forked by developers.

2.1.7 File repository forking

A file repository fork is mainly used to make contributions to original repositories and is beneficial for the OS software community [6]. Motivating reasons for developers to fork repositories include submitting pull requests, fixing bugs, adding new features and keeping copies. A repository written in a developer's preferred programming language is more likely to be forked and developers mostly fork repositories from reliable creators. Attractive repository owners include organisations, as they have more followers.

2.2 Forking patterns

Regardless of forking type, there are three forking patterns that can be identified in GitHub: single, or once only; intermittent; and steady. A single fork pattern refers to developers who fork programming language repository files once a month and then not at all in consecutive months. Intermittent refers to forking over some months, then not in others, then again in later months. A steady fork pattern refers forking files consistently for a defined period, such as every month for 12 months (Table I).

TABLE I. FORK PATTERN

Respository	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	Fork Pattern
Rick/dotfiles	1	0	0	0	0	0	0	0	0	0	0	0	Fork Once Only
Droogans/unmaintainable-code	4	0	4	2	4	0	2	69	3	3	2	2	Fork Intermittent
Electron/electron	287	260	266	198	229	225	191	190	164	223	183	175	Fork Steady

3. SOFTWARE SURVIVAL AND PROGRAMMING LANGUAGE SURVIVAL IMPORTANCE

Many critical factors have been discussed in the literature on success of closed source language development [23] but a focus on programming language assessment must continue in the new OS software development culture. Not only can we expect voluminous source codes to be contributed by developers but also an increase of new OS programming languages added, driving competition for programming languages survival.

The survival of a programming language is critical to a repository and a project owner, as a language without forking is equivalent to no new source code, implying no development, potentially as the chosen programming language failed to produce source code that was ready in time to develop and deliver a software product. In other words, it is difficult to develop a programming language used in a repository file quickly and submit it to a production environment. The longer a repository file remains in GitHub without developer interest, the greater the likelihood of termination once the public repository file expires. It is therefore a waste of development time and effort to create that repository file.

A surviving programming language is one that is more open to interoperability and integration to build ecosystems and emerging technology agility and mobility. A surviving programming language can also reduce the risk of replacing another programming language and developing other components.

Ranking of popular or sustainable programming languages is one way to assure developers a language is reliable to adopt. Unfortunately, however, programming language popularity, sustainability and successability assessments vary across companies, projects, platforms to platform, and communities [6,14], making comparing results difficult.

There is no one method to assess programming language popularity and rank importance regardless of how the language used and adopted. There is limited literature on assessing programming language popularity, success and sustainability by measuring fork performance as, to date, forking has not been instrumental as a viable process for time to production on repository files.

Since forking forms an integral part of OS software development, ranking importance of programming languages is relevant. A programming language forking rank result could be of benefit when considering and selecting the right programming language to adopt, use and fork in a platform, and obtain the right community support. However, ranking programming language forking success or popularity is not an easy task. There are a number of factors to consider, including the target platform, elasticity of a programming language, topic of interest, time to production, programming language fork performance, and community support. Most importantly, both forking and programming language are time-independent and assessing them can be daunting as forking fluctuates inconsistently.

4. SURVIVABILITY PREDICTION USING THE K NEAREST NEIHBOUR METHOD

The K-Nearest Neighbour (KNN) method is one of the most popular non-parametric classification algorithms because it is simple, effective, and more accurate than many other classification algorithms [22,24–28]. The KNN

method is used in data mining and statistics because of its simple implementation and significant classification performance produces more accurate results than many other algorithms [25,29].

It was first introduced in 1951 by Fix and Hodges in their unpublished report for the US Air Force School of Aviation Medicine. In 1967 Cover and Hart formalised the original idea and discovered the main properties of this method [22]. The KNN method can also handle mixed Euclidean distance, adopted in this paper.

We aimed to predict the lifespan of a programming language through forking using KNN, given forks range from a few months to several months. The algorithm calculates Euclidean distance from the 12 months of the forking period based on the forking pattern categories to evaluate which types of programming language repository file are short- or long-lived by the minimum distance. We chose a 12-month period rather than days or weeks as we did not find a significant number of forked changes on repositories over the shorter time-frame. We used the three patterns defined above: single, intermittent and steady.

According to the Euclidean distance formula, the distance between two points in a plane with coordinates (x, y) and (a, b) is given by

$$dist((x, y), (a, b)) = \sqrt{(x - a)^2 + (y - b)^2}$$

and the a, b, x and y variables must be numeric. As such, we converted non-numeric variables from a forking dataset downloaded from GitHub (January–December 2017) into numeric variables (Table II). We adopted one of the queries from [30] into the Google Big Query using the Select Statement (see below, highlighted the condition to retrieve only created forked repositories. We downloaded 1,000 repository files from GitHub, randomly categorised them alphabetically.

```
SELECT events.repo.name AS events_repo_name,
COUNT(DISTINCT events.actor.id) AS events_actor_count
FROM (SELECT * FROM TABLE_DATE_RANGE
([githubarchive:day.],TIMESTAMP('2017-01-
01'),TIMESTAMP('2017-12-31'))) AS events
WHERE events.type = 'ForkEvent'
```

TABLE II. VARIABLES DEFINED FOR PROGRAMMING LANGUAGE SURVIVABILITY

Name	Description	Source	Variable	Type[a]	Binary
Events_repo_name	A repository file name	GitHub	x1	C	N/A
Repo_type	Own creation (from the description of the source code link)	NA	x2	C	N/A
Prog Lang Name	Programming Language 1. Python, 2. C++, 3. Java, 4. C, 5. C#, 6. PHP, 7. R, 8. JavaScript, 9. Go, 10. Assembly	GitHub [16]	x3..x13	C	1 Yes, 0 No
Open Source recognised license	Official Open Source license : BSD 3-Clause "New" or "Revised" license,BSD 2-Clause "Simplified" or "FreeBSD" license,GNU General Public License (GPL),GNU Library or "Lesser" General Public License (LGPL),MIT license,Mozilla,Common Development and Distribution License (CCDL) Public License 2.0,Eclipse Public License	[15]	x14..x21	C	1 Yes, 0 No
Open source technology	Open Source Technology refers to OpenStack, Progressive Web Apps, Rust, R, the cognitive cloud, artificial intelligence (AI), the Internet of Things	[14]	x22..x32	C	1 Yes, 0 No
Fork 12 surviving months	Fork detected every month from Jan to Dec	NA	x33	B	1 Yes, 0 No
Environment compliance	satisfy environment compliance (sustainable top 10 programming language, recongised license, open source technology	NA	x34	B	1 Yes, 0 No
Forking month	January to December	NA	x35..x47	N	N/A

[a] Binary, B; Character, C; Numeric, N.

To satisfy environment compliance around product, programming language and license, we referenced Open Source Technology's list of top products developers are interested in [31] and top officially recognised OS licenses [32], and IEEE's top programming languages and licenses adopted in OS repository files [33], namely Python, C, Java, C, C#, PHP, R, JavaScript, Go and Assembly.

A repository file name is a name given to uniquely identify a piece of source code that stored in the GitHub. Due to some filenames non-interpretable, the conversion from characters into binary is difficult. For instance a repository file name in GitHub labelled as "1ppm/1ppmLog". For all variables except the file repository file name, attributes with text characters were converted into binary numbers (1=yes, 0=no); e.g., programming language names, repository file and license. For instance, for the programming language JavaScript, 1 indicated JavaScript was used and 0 indicated it was not JavaScript.

In total, there were 47 attributes, with two added to determine duration of programming language repository file survival (in months) and how many repository files complied with the criteria published in [31–32].

Our definition of a long-lived programming language repository file was based on detecting a consecutive 12-month forking performance; short-lived was no fork counts detected in the 12-month period. For example, a JavaScript social media repository file was predicted to have a short-lived outcome as there was no fork in the 12-month period versus a Python machine learning repository file was predicted to be long-lived, having visible monthly forking.

In total, 47,000 forking data over the 12-month period were evaluated for Euclidean distance, using the three patterns, to determine which programming language repository files were short- or long-lived by the minimum distance.

5. PROGRAMMING LANGUAGE REPOSITORY FILE CATEGORISATION AND FORK PATTERN CLASSIFIERS

Categorising the forking dataset into single, intermittent, or steady patterns revealed nine types of programming language repository files based on environment compliance and fork performance (Table III).

TABLE III. FORKING PATTERNS

Forking Pattern	**Programming Language Repository Files**
Once Only	Specific Respository File (SPF)
Intermittent	Specific repository file met official licence compliance and adopted a modern sustainable programming language (SRFMSPL)
	Specific repository file met official licence compliance (SRFOL)
	Specific repository file met official licence adopted a traditional sustainable programming language (SRFOLTSPL)
	Specific repository file adopted a traditional sustainable programming language (SRFTSPL)
Steady	Specific repository file that did not meet the full environment licence but has healthy fork (SRFHF)
	Specific repository file met official licence compliance that has healthy fork (SRFOLHF)
	Specific repository file met official licence compliance and adopted a modern sustainable programming Language that has healthy fork (SRFOLMSPLHF)
	Specific repository file adopted a traditional sustainable programming language that has healthy fork (SRFTSPLHF)

TABLE IV. CATEGORISING PROGRAMMING LANGUAGE RESPOSITORIY FILE FORKS AS SHORT- OR LONG-LIVED

Short-lived		**Long-lived**	
Abbreviation	**#**	***Abbreviation***	**#**
SPF	138	SRFOLHF	32
SRFOL	104	SRFHF	20
SRFTSPL	172	SRFTSPLHF	42
SRFOLTSPL	347	SRFOLMSPLHF	5
SRFMSPL	5	SRFOLTSPLHF	107
SRFOLMSPLHF	28		
Total	794	Total	206

Fig. 1. Categorising programming language repository file forks as short- or long-lived.

6. CLASSIFIER RESULTS

The results of using Euclidian distance to categorise the programing language repository file forks are shown in Table IV and Figure 1. A high number were short-lived (79.4%) and only a small number were long-lived (20.6%).

Our results identified some non-sustainable programming languages that lacked environment compliance survived as long as sustainable programming languages that met environment compliance: 94/206 repository files did not completely meet environment compliance but survived well, e.g., CSS, Kotlin, Emacs Lisp and Jupiter Notebook. Long fork survival could be due to a developer community supporting an OS technology trend; e.g., machine learning, web applications or android operating systems.

We found the majority of sustainable programming languages were short-lived because of low or no license compliance. The data revealed many developers chose a low compliance license – development mountain copyright, CC BY NC SA 4.0, Creative Commons Attribution 4.1, WTFPL, or Educational Content License – however, as these licenses are less popular and/or have low compliance some developers are hesitant to contribute [32]. In contrast, long-lived programming language repository files aligned with the top 10 sustainable programming languages [33]. Nevertheless, some repository files that adopted programming languages not in the top 10 – such as Python, PHP, Swift, Shell and Ruby – also survived well.

6. K-NEAREST NEIGHBOUR RESULTS

The results of the KNN method are summarised in Table V, shows the classification of programming language repository files by KNN/ Euclidean Distance, and illustrated with four case studies below.

TABLE V. CATEGORISING PROGRAMMING LANGUAGE REPOSITORY FILES SORTED BY EUCLIDEAN DISTANCE

Classification	File Count	Euclidean Distance	Rank
SRFOTLSPLHF/ SRFOLMSPLHF	113	0	1
SRFTSPLHF	41	1	109
SRFOLHF/ SRFOTLSPLHF	33	1	113
SRFOLSPL/ SRFOLMSPL	374	1.4	187
SRFHF	20	2	562
SRF	1	3.2	566
SRFSPL/SRFOL	281	2.2	582
SRF	137	3.2	863

TABLE VI. ENVIRONMENT COMPLIANCE

Full compliance	Long-lived	Short-lived	Total
Yes	111 (TP)	94 (FP)	205
No	0 (FN)	795 (TN)	795
	111	889	1,000

6.1 Case One

A programming language repository file is found to associate with the following properties: one of the top ten OS technologies [31], met legitimate license compliance [32], adopted a sustainable programming language [33], and displayed monthly forking over the last 12 months. This file is predicted to be a long-lived surviving programming language file with healthy forking. Our results predict SRFOTLSPLHF or SRFMTLSPLHF would fall under this category.

6.2 Case Two

A programming language repository file is found not to associate with one of the following properties: one of the top ten OS technologies [31], met legitimate license compliance [32], adopted a sustainable programming language [33], and displayed monthly forking over the last 12 months. This file is predicted to be a long-lived surviving programming language file with healthy forking. Our results predict SRFHF would fall under this category.

6.3 Case Three

A programming language repository file is found not to associate with more than one of the following properties: one of the top ten OS technologies [31], met legitimate license compliance [32], adopted a sustainable programming language [33], and did not display monthly forking over the last 12 months. This file is predicted to be a lower surviving programming language. Our results predict SRF would fall under this category.

6.4 Case Four

A programming language repository file is found not to associate with more than one of the following properties: one of the top ten OS technologies [31], met legitimate license compliance [32], adopted a sustainable programming language [33], but displayed monthly forking over the last 12 months. This file is predicted to be a lower surviving programming language. Our results predict SRFOL would fall under this category.

7. EVALUATION

In this paper, we proposed evaluating sensitivity and specificity to describe test performance, as these parameters remain true regardless of the population of programming language repository files to which the test is applied.

Definitions of environment compliance parameters are presented in Table VI, where: true positive (TP) is the number of programming language repository files that met environment compliance and were classified as long-lived; false positive (FP) is the number that met environment compliance and were mistakenly classified as short-lived; true negative (TN) is the number that did not meet environment compliance and were classified as long-lived; and false negative (FN) is the number that did not meet environment compliance and were mistakenly classified as short-lived.

For this study, we divided the data into training (80%) and testing (20%) samples. We used the KNN method to classify the class of the repository files then calculated the Euclidean distance between the forking period and forking pattern. After determining the parameter *k* and running the KNN algorithm, accuracy was calculated using sensitivity, specificity and precision. The formulas of the four measures are outlined below.

Accuracy refers to the proportion of true results among the total number of positive and negative cases examined.

$$Accuracy = TP+TN/(TP+TN+FP+FN)$$

For this study, accuracy is 111+795/(111+795+94+0)= 0.906 (90.6%).

Sensitivity is the proportion of long-lived programming language repository files that meet full environment compliance, and specificity is the proportion of short-lived programming language repository files that meet full environment compliance. Hence, the formula is

$$Sensitivity = TP/TP+FN$$

$$Specificity = TN/TN+FP$$

For this study, sensitivity is 111/111+0=1 (1%) and specificity is 795/795+94=0.894 (89.4%).

Precision is the ratio of correctly predicted positive observations to the total predicted positive observations; that is, of all programming repository files that appeared to survive, how many actually survived? High precision therefore relates to a low false positive rate.

$$Precision = TP/TP+FP$$

For this study, precision is 111/111+94=0.542 (54.2%).

Figure 2 summarises all four metrics.

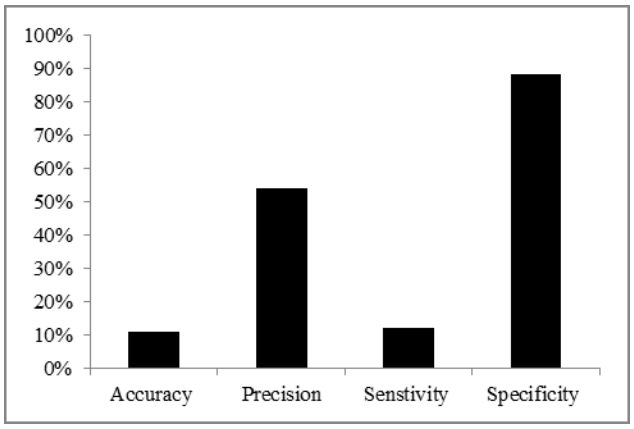

Fig. 2. Evaluative results comparison of the dataset.

8. DISCUSSION

Figure 1 highlights that there are less long-lived programming language repository files than short-lived. For a programming language repository file to survive it must satisfy environment compliance properties; the data reveal most do not comply and are therefore short-lived. In other words, many project developers or owners who created repository files may have ignored, or failed to pay attention to, environment compliance factors, such as technology trends and licensing.

Table VI is a statistical overview of programming language repository file lifespan and Figure 2 is an overview of the test result accuracy showing a breakdown of accuracy, sensitivity, specificity and precision. Our findings reveal that it is necessary for developers to pay attention to environment compliance before developing a repository file if they want to ensure healthy forking and file survivability.

The predictive results help us to better categorise developers' motivations for forking. The existing literature identified seven categories of forking: OS, project, software, social, code, programming language and repository. Our data show long-lived forked programming language repositories that satisfy environment compliance are potentially related to social, programming language and repository forking. In contrast, short-lived forked programming languages that are environment compliant are related to code, OS and project forking.

Our future work in this area will focus on introducing new environment compliance variables to fast-growing project code that is forked from very large-scale programming languages with boundary conditions. In addition, we will evaluate which machine learning method can accurately and reliably predict fork patterns for short-lived and long-lived programming languages.

REFERENCES

1. L Nyman. 2014. Hackers on forking. *Proceedings of the International Symposium on Open Collaboration, ACM,* New York, NY, USA, ISBN: 978-1-4503 30169. 4–12.
2. L Nyman, & T Mikkonen. 2011. To fork or not to fork: fork motivations in SourceForge projects. *International Journal of Open Source Software & Processes 3*(3), 1–9.
3. L Nyman, T Mikkonen, J Lindman, & M Fougère. 2012. Perspective on code forking and sustainability in open source software. *Proceedings of the IFIP International Conference on Open Source Systems,* Open Source Systems: Long-Term Sustainability, pp. 274–279. Buenos Aires, Argentina.
4. F Ikuine, & H Fujita. 2014. How to avoid fork: the guardians of Denshin 8 Go, Japan. *Annals of Business Administrative Science 13,* 283–298.
5. KH Fung, A Aurum, & D Tang. 2012. Social forking in open source software: an empirical study. *CAiSE Forum* 50–57.
6. F Tegawendé, T Bissyandé, F Thung, D Lo, LX Jiang, & L Réveillère. 2013. Popularity, intero-perability, and impact of programming languages in 100,000 open source projects. *Proceedings of the Computer Software and Applications Conference (COMPSAC), IEEE 37th Annual Conference,* Kyoto, Japan.
7. Ray, B. and Kim, M. 2012. "A Case Study of Cross-System Porting in Forked Project", in Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundation of Software Engineering
8. Ray, B., Posnett, D., Filkov, V., and Devanbu, P. 2014. "A Large Scale Study of Programming Languages and Code Quality in Github", Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, November 16-21, 2014, Hong Kong, China
9. AE Azarbakht., & C Jensen. 2017. Longitudinal analysis of the run-up to a decision to break-up (fork) in a community. *Proceedings of the IFIP International Conference on Open Source Systems.* OSS 2017, Springer.
10. M Biazzini, & B Baudry. 2014. "May the fork be with you": novel metrics to analyze collaboration on GitHub. *Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics,* June 2014, Hyderabad, India.
11. B Chua. 2015. Detecting sustainable programming languages through forking on open source projects for survivability. Proceedings of the *IEEE* International Symposium on Software Reliability Engineering (ISSRE) 2015 in conjunction with a WOSAR workshop, IEEE, Gaithersburg, USA. 120–124.
12. B Chua. 2017. A survey paper on open source forking motivation reasons and challenges. *Proceedings of the Pacific Asia Conference of Information Systems (PACIS)*, Malaysia, Langakawi.
13. L Dabbish, C Stuart, J Tsay, & J Herbsleb. 2012. Social coding in GitHub: transparency and collaboration in an open software repository. *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work* Seattle, Washington, USA.
14. J Jiang, D Lo, JJ He, X Xia, PK Singh, & L Zhang. 2016. Why and how developers fork what from whom in GitHub. *Journal of Empirical Software Engineering 100*(21), 1–32.
15. VG Krogh, S Haefliger, S Spaeth, & MW Wallin. 2012. Carrots and rainbows: motivation and social practice in open source software development. *Journal of MIS Quarterly 362*, 649–676.
16. G Robles, & M Gonzalez-Barahona. 2012. A comprehensive study of software forks: dates, reasons and outcomes. Open

source systems: long-term sustainability. *IFIP Advances in Information & Communication Technology 378*(1), 1–14.
17. B Chua. (under review). Applying systematic literature review and content analysis methods to analyse open source developers' forking motivation interpretation, category and consequences. *Journal of Australian Information Systems*.
18. J Biolchini, P Mian, A Natali, & G Travassos. 2005. *Systematic Review in Software Engineering*. Technical Report RT-ES 679/05, COPPE/UFRJ, Rio de Janeiro, Brazil.
19. B Kitchenham, OP Brereton, D Budgen, M Turner, J Bailey, & S Linkman. 2009. Systematic literature reviews in software engineering – A systematic literature review. *Journal of Information and Software 51*(1), 7–15.
20. S Cavanagh. 1997. Content analysis: concepts, methods and applications. *Nurse Researcher 4*(3, 5–16.
21. H Hsiu-Fang, & SE Shannon. 2016. Three approaches to qualitative content analysis. *Journal of Qualitative Health Research 15*(9), 1277–1288.
22. T Cover, & P Hart. 1967. Nearest neighbour pattern classification. *IEEE Transactions on Information Theory 13*(1), 21–27.
23. Linberg, K.R., 1999. Software developer perceptions about software project failure: a case study. *Journal of Systems and Software 49*(2), 177–192.
24. J Gou, L Du, Y Zhang, & T Xiong. 2012 A new distance weighted k-Nearest Neighbor classifier. *Journal of Information and Computer Sciences 9*(6), 1429–1436.
25. D Hand, H Mannila, P Smyth. 2001. *Principles of Data Mining.* MIT Press, Cambridge.
26. K Odajima, & AP Pawlovsky. 2014 A detailed description of the use of the kNN method for breast cancer diagnosis. In: *Biomedical Engineering and Informatics, 7th International Conference. IEEE*; May 2014, 688–692.
27. H Wang. 2002 *Nearest Neighbours without k: A Classification Formalism based on Probability.* Technical report, Faculty of Informatics, University of Ulster, N. Ireland, UK.
28. X Wu, V Kumar, JR Quinlan, J Ghosh, Q Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P.S. Yu, Z.H. Zhou M.Steinbach, D.J. Hand and D.Steinberg.2008 Top 10 algorithms in data mining. *Knowledge Information Systems 14*(1), 1–14
29. F Sebastiani. 2002 Machine learning in automated text categorization. ACM Computing Surveys 34(1), 1–47
30. https://octoverse.github.com/. Accessed 16th Feb 2019.
31. https://opensource.com/article/17/11/10-open-source-technology-trends-2018. Accessed 5th Jan 2018
32. https://opensource.org/licences. Accessed 5th Jan 2018
33. https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018. Accessed 5th Jan 2018.