

Investigating Evolution in Open Source Software

Jordan McDonald and Des Greer(✉)

Queen's University Belfast, Belfast BT7 1NN, UK
{jmcdonald23,des.greer}@qub.ac.uk

Abstract. Lehman's well-known laws of software evolution have existed since the early 1980's and although they have been nuanced, augmented and discussed many times since then, software and software development practices have changed dramatically since then, not least due to the rise and popularity of open source software (OSS). OSS is written collaboratively with the process and products publically observable, whereas the original laws were derived based on a very different context. The question then arises if Lehman's laws apply to modern day OSS software. The GitHub repository is the most comprehensive source of OSS projects and is used here to obtain data on how OSS projects have evolved. This work uses one hundred open source projects hosted on GitHub. Metrics are obtained via the provided API, using a purpose-built workbench and several of Lehman's laws are evaluated using the data available. Coupled with a critique of how judgements can be made from the data available, the study has discovered that the evidence does not support many of the laws. An important proviso with such an approach is the limitation on what data can be extracted and/or inferred from the GitHub API. Nonetheless, there is enough of a challenge made to the laws to warrant further study and a need to revisit some of the laws in the context of open source development.

Keywords: Open source software · Software evolution · Repository mining · Empirical software engineering

1 Introduction

Evolution of Open Source Software products is very different from closed source projects yet almost all investigations leading to Lehman's laws of Software Evolution has been based on closed-source projects [1, 2]. Thus there is scope for re-investigating the applicability of these laws in modern open source software projects.

Lehman's first three laws of software evolution were formulated in the mid-1970s arising from a study on IBM data from 1968. Further laws were added in 1980, 1991 and 1996 [3, 4]. Collectively the laws attempt to describe and explain the factors that drive growth and development of software as well as the forces that lead to reduced progress. The goal of this paper is to examine these laws in the context of modern open source projects hosted on GitHub, with a dataset extracted via the GitHub API. GitHub is a hosting service designed for collaboration on a centralized repository of source code. Any user of the website can 'Clone' any public repository and read or alter the code, this serves as the backbone of modern open source development and helps facilitate the 'fork

© Springer Nature Switzerland AG 2019
S. Misra et al. (Eds.): ICCSA 2019, LNCS 11623, pp. 242–256, 2019.
https://doi.org/10.1007/978-3-030-24308-1_20

and pull' model of development. In addition to code hosting, collaborative code review, and integrated issue tracking, GitHub has integrated social features. Users are able to subscribe to information by "watching" projects and "following" users, resulting in a feed of information on those projects and users of interest [5].

At the time of writing, GitHub claims to host over 100 million projects with 31 million developers [6] which would make it the largest host of source code in the world. This coupled with the easily accessible GitHub API, which can be used to obtain data on commits, code churn, issues, watchers and pulls, among other metrics, means that GitHub provides an obvious foundation to examine Lehman's laws as they apply to Open Source Software (OSS).

This paper will perform a large scale analysis of open source projects hosted on GitHub, extracting data at the repository level in order to determine how well the selected Lehman's laws hold. The next section will provide some background to the study, Sect. 2 will describe the background to the work, Sect. 3 will describe the methodological aspects of the study. Section 4 will discuss the results of the work while Sect. 5 will discuss the validity of the study. Finally, in Sect. 6 we provide attempt to draw conclusions from the work and identify possible further work.

2 Background and Related Work

2.1 Laws of Evolution

Lehman's laws have undergone multiple changes. Lehman originally qualified the application of such laws by distinguishing between three categories of software – S-programs, P-programs and E-Programs [3]. In this paper we will limit the scope to E-programs, the reasonable assumption being that the collaborative projects selected will be of this type i.e. broadly "computer applications in the *real world"* [4]. There is, of course, a small probability that sampled GitHub projects will be S or P-type but both of these types are less obvious candidates for collaboration via GitHub due to the static nature of the problems they solve [7]. Table 1 provides a summary of the relevant laws as they apply to E-type systems. Laws 4, 5, and 8 do not lend themselves to this type of study and so are not included.

Table 1. Lehman's laws of software evolution (Greyed not studied).

	Law description
1	(1974) "**Continuing Change**" - a system must be continually adapted or it becomes progressively less satisfactory
2	(1974) "**Increasing Complexity**" - as a system evolves, its complexity increases unless work is done to maintain or reduce it
3	(1974) "**Self-Regulation**" - system evolution processes are self-regulating with the distribution of product and process measures close to normal
6	(1991) "**Continuing Growth**" - the functional content of a system must be continually increased to maintain user satisfaction over its lifetime
7	(1996) "**Declining Quality**" - the quality of a system will appear to be declining unless it is rigorously maintained and adapted to operational environment changes

2.2 Related Work

Attempts at general data mining from GitHub have been prominent in recent years, Kalliamvakou et al. [5] focused on avoiding common pitfalls in GitHub mining and concluded that there is valuable data to be found if these are avoided. More generally, there is a systematic literature review on the evolution of open source projects, where the authors examine the data sets utilised, sources of the data and research trends in recent years [8]. The authors found that Lehman's laws do not hold in certain cases, with individual laws in the research yielding contradicting results for OSS projects.

Yu and Mishra [9] investigated the quality of evolving software by mining bug reports and provide evidence to confirm law seven. Sheoran et al. [10] investigated "watchers" on GitHub. The paper looks at the contributors to a project, tracking the process of a user becoming a watcher to finally contributing to a project, finding that this process accounts for a huge bulk of the tested projects eventual contributors. Another study on this topic was conducted by Ben et al. [11] which performed visualisation on metrics related to commits, low level code statistics and lines of code on a single project, but this restriction limits the generality of the research. Gousios et al. [12] look in depth at the GitHub 'fork and pull' model of development on a sample of 291 projects. The metrics utilised are among the widest ranging, considering feature sets for the pull request itself, the project and the developers involved. An analysis was made on what projects utilise this model, the turnover rate of pull request and why requests are rejected. Borges et al. [13] provide insight into what constitutes project popularity on GitHub making use of the starring mechanism. Bissyandé [14] analyses issues as part of open source software, correlating the data with watchers, forks and other metrics.

A similar study to that presented in this paper was conducted on a long running FLOSS project, glibc inside a SCM repository with over 20 years of history [7]. The paper also centres on Lehman's laws and makes use of commits, lines of code and files changed to represent evolution. A downside to this study is single project focus. A nine project study is used in [15] with a focus on long running projects. The findings, based on code level metrics only confirmed the laws of continuing change and continuing growth for all programs but gave evidence that the other six laws were sometimes violated. Skoulis et al. [16] also delves into software evolution and Lehman's laws, however from the context of databases. Rather than confirm or challenge the laws they offer general observations on the evolution process and its influences. Work also exists looking at the evolution of the Linux kernel [17]. This study found that several of Lehman's laws were supported.

There has also been an extensive study of the Eclipse development environment by Mens et al. [18]. They analysed the source code of Eclipse, for example capturing differences between releases and relating these with various quality measurements. They found strong support for laws one and six but only partial support for law two. This support for laws one and six from Eclipse data is repeated in [19], as well as qualified support for law eight but not for the other laws, where they state that the evidence is contradictory.

It also important to consider relevant research that focuses on the laws outside of the given OSS context. There is some evidence to support Lehman's laws in a

refactoring based development approach [20]. This study uses a single project case study and concludes that the laws apply in this context.

Prior studies that are similar to the approach in this paper have limitations such being based on a single (or just a few) project or being very specific in scope. By contrast this study will use a large data set with variation in the language.

3 Methodology

3.1 Research Questions and Hypotheses

In order to provide scope and direction to the research we have defined research questions with multiple hypotheses that will attempt to draw out appropriate relationships for each law with the metrics extracted via the GitHub API. Each (alternative and null) hypothesis is also accompanied by justification and reasoning which explain the choice of metric used for each law. Inevitably, we will have to resort to proxy measures since many of the concepts in the laws are not directly measureable. Hence for each hypothesis we will give a justification for the metric(s) used. Indeed several of the laws are difficult to assess from solely GitHub data. In that case we have omitted to study laws 4, 5 and 8.

RQ – Is it possible using data extracted from the GitHub API to determine if OS software evolution over time reflects Lehman's laws?
H_1a – If the number of commits decreases the number of star gazers will also reduce (laws 1 & 6).
H_10 – There is no relationship between the number of stargazers and the number of commits.
Justification – In this particular case a caveat should be noted. Stargazer count is a reflection of developers rather than users (GitHub is not designed for end users). However, OSS developers typically exists as a result of user demand. Indeed it is also often the case that an OSS user is also a developer. The assumption can be made that if a developer stops stargazing the software has become less satisfactory. Commits can be used to represent changes made to the code.
H_2a – Total lines of code increases as a software system evolves (law 2).
H_20 – Total lines of code will not increase as a software system evolves.
Justification – To represent complexity LOC provides a convenient measure. There is good support in the literature [21] to show that LOC and McCabe's Cyclomatic Complexity have a stable linear relationship. Extending to use other complexity measures remains in plans for future work.
H_3a – Issues, additions and deletions over time for are normally distributed (law 3).
H_30 – Issues, additions and deletions over time are not normally distributed.
Justification – To capture the essence of 'self-regulation', issue counts were chosen as a suitable metric. This represents how 'processes' are managed by the developers and forms the focal point for communication and decision making on code changes. The product of software evolution can be assessed via code output, at the lower level of addition and deletion counts.
H_4a – Project issues increase as code churn decreases (law 7).

H_40 – There is no relationship between project issues and code churn.
Justification – To capture the declining quality of a software system, issues is an appropriate measure. As quality declines it seems intuitive that more issues will be created by developers. Code churn will be represented by LOC, which satisfies the need for rigorous maintenance to prevent a decline in quality.

3.2 Project Selection

Figure 1 demonstrates the project selection process. The ten languages have been chosen based on a ranking system [22]. These derived (based on total active repositories, both public and private but excluding forks) as of August 2015: JavaScript, Java, Ruby, PHP, Python, CSS, C++, C#, C, and HTML. GitHub's advanced search facility allows descending ordering of the 'most stars' for a programming language. Each sequential project is then evaluated against two criteria.

a. Duration of project life on GitHub, with a five year threshold, arbitrarily chosen to ensure evolution can be mapped over a sustained period of time.
b. It is very common for projects to use multiple programming languages. However GitHub allows users to examine a project for the breakdown of languages utilised. Using this, each project had to meet a 50% target language affinity requirement. This process was applied to two hundred projects in total, the final dataset of one hundred was then randomly selected with the intent of taking ten projects from each programming languages group of twenty.

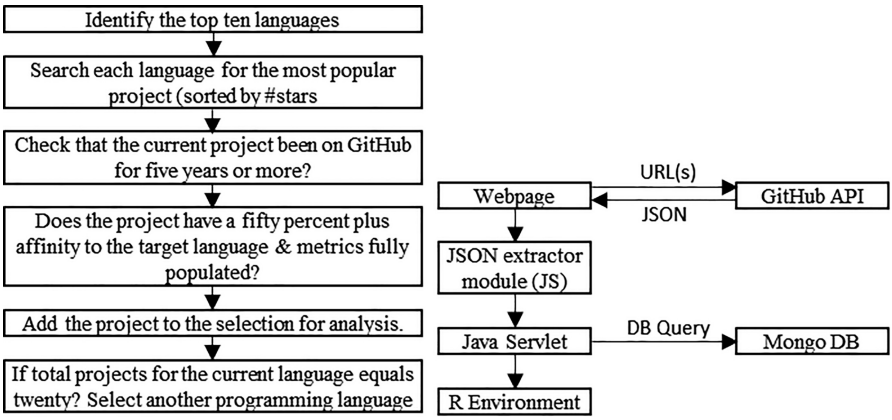

Fig. 1. GitHub project selection process and GitHub extraction workbench architecture

3.3 Data Collection

GitHub provides a robust API which is ideal for mining the data associated with a project. The current version of the API is version 3, and all requests are performed over HTTPS, the data is returned in a JSON format which allows simple parsing of the required metric. A workbench (Fig. 2) has been devised which will handle the

automated collection of the data for each of the one hundred projects and to execute statistical functions. The method utilised to collect GitHub data used AJAX as implemented in the JQuery JavaScript library, and then once processed data was stored in a MongoDB database. In order to test hypotheses, various statistical methods are provided via the R environment [23] which was integrated into the workbench.

3.4 Data Analysis

Each metric is associated with an accompanying time series that signifies the start of a weekly interval. The dataset itself is organised into a vector with each point containing weekly counts of the frequency of the metric in that particular time period. Some data points may have gaps between frequencies that exceed the weekly structure, therefore padding has been introduced to fill the gaps in a project as required, in this case each padded weekly interval will be assigned a zero to signify no activity in that period. To ensure the integrity of the research the first six months for each projects have been 'trimmed'. This is reasonable since many projects that have origins that predate the GitHub platform. This also removes the possibility of initial 'dump' of data from a pre-existing source thus polluting the results with the potential for significant statistical outliers. The metrics that will be extracted from the API in order to quantify the analysis are listed below.

- Stargazers - Repository Starring is a feature that lets users bookmark repositories. Stars are shown next to repositories to show an approximate level of interest.
- Commits - A commit, or "revision", is an individual change to a file or set of files.
- Additions & Deletions
- Issues - Issues are suggested improvements, tasks or questions related to the repository.
- Issues Comments – Messages that a user has attached to a specific issue.
- LOC – total lines of code at a certain time point.
- Growth Rate – how much a metric changed per time interval.

3.5 Statistical Analysis

Growth Rate. This concept has significant relevance. We use it in tandem with an LOC metric to test H_2. We can express growth rate and average growth rates as percentages.

Growth Rate - $G = ((X - Y) \div Y) \times 100$

Average Growth Rate - $G = ((X \div Y)^{1 \div n} - 1) \times 100$ where X = current value, Y = past value, n = total samples

Shapiro Wilk Test. This particular test will be applied to the three metrics stated for H_3, in order test for normality. As per tradition, we set alpha at a 95% confidence level i.e. alpha = 0.05.

Cross Correlation. To adequately test H_1 and H_4 a cross correlation will be performed which will quantify the relationship between two time series by identifying lags of series x that will be useful predictors of series y. In the case of this research, multiple lag values will be considered to determine if a change in one metric weeks prior will

have an impact on a series weeks in the future, in other words to determine if x leads y. We have used the R package cross correlation function *ccf*(…) [23].

4 Results and Discussion

4.1 H_1a: If the Number of Commits Decreases the Number of Star Gazers Will also Reduce

A lagged cross correlation was performed with multiple different values in order to determine if and when the impact of making a change i.e. a commit will have a direct effect on the number of stargazers, and in particular what duration after a commit the change is felt most significantly. The results of this experiment are shown in Table 2 for a lag from −9 to 0 weeks.

The results show a clear relationship between the amount of the lag applied to the commits and the percentage of positive correlations that have been attained between the lagged commit count and the present stargazer count. As the lag is increased (in this context each increment represents the count of commits a week further into the past) the amount of correlation begins to decrease which indicates that the further apart the commit frequency in a particular week from the present stargazer count, the less impact it will have on the amount of stargazers. It is possible that in the case of extreme lag applied that the effect of that change has already been felt at some point in the interim, therefore it may have already changed the count of the stargazers in a positive or negative way. If we now consider the inverse of this trend it appears that if changes in the amount of commits contributed to the project are recent (0 lag to −4 lag) the amount of stargazers is more likely to correlate which would suggest that the amount of commits made recently has a greater bearing on the number of stargazers than those which typically happened over a month prior. If we consider this from a potential stargazer's point of view it stands to reason that they will be more likely to 'star' or 'unstar' the project based upon the recent changes that have been made to the system rather than those that happened in points in time beyond a few weeks due to having a greater investment in commits that have more immediate effects on the project.

Table 2. % Positive cross correlation at different lag intervals for #Commits and #Stargazers

Lag amount	% Positive correlations between #commits and #stargazers
0	60%
−1	61%
−2	57%
−3	60%
−4	60%
−5	54%
−6	55%
−7	55%
−8	50%
−9	51%

Regarding accepting or discarding the null hypothesis, the value in the case of all lags is not conclusive enough to be able to determine this. An argument could be made that the lesser lag values support the hypothesis. In particular, the −1 commit lag which is the best performing correlation percentage with stargazers indicates that the optimum time is week before the stargazers react to the commit count and decide whether to remain stargazers or to stop following the project. Figure 2 visualises the distribution of correlation values at different lag intervals, highlighting the almost random nature obtained, thus supporting the null hypothesis. Table 2 shows the mean correlation values for each lag interval and while all remain positive the significance of lag interval is evidently minor. Therefore, we cannot reject the null hypothesis.

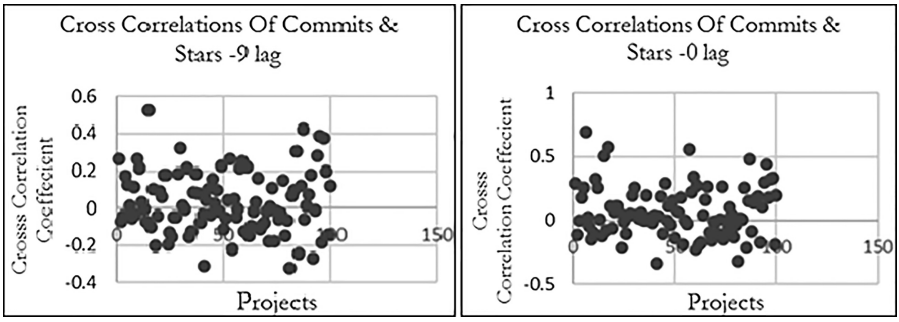

Fig. 2. Distribution of correlation for the two extreme lag intervals

Law one and six both state that in order to maintain user satisfaction the project will need to continually change and grow to maintain user satisfaction. A reason why this does not apply to the context of the GitHub platform could be attributed to the starring process which serves as a repository 'bookmark' for the user to show an level of interest that does not extend to receiving notifications etc. about the project. This would suggest that independent of the amount of commits (change) made the user will continue to remain starred until they have a reason to change that stance (become less satisfied)/stop supporting the project which highlights a clear disconnect between these particular laws and the GitHub platform (Table 3).

4.2 H_2a – Total Lines of Code Increases as a Software System Evolves

LOC metrics were organised into a vector and a growth rate algorithm applied to determine the average percentage growth for each week from the first and last week's LOC total. Thus from 100 projects, 100 growth rate values were generated, allowing the assessment of the percentage of projects that increased in size over time. Figure 3 visualises the results of this process. The majority of the projects increase in size as the software system evolves. This is generally to be expected since as time progresses the demand for new functionality is intuitively expected and will be constant in order to support the user base. However, there remain several projects that produced contrary results and reduced in size. Law two states that this could be the side effect of work

Table 3. Mean cross correlation for stargazers and commits organised by interval

Lag interval	Mean cross correlation for #commits and #stargazers
0	0.0321
−1	0.0258
−2	0.0266
−3	0.0307
−4	0.0336
−5	0.0381
−6	0.0461
−7	0.0367
−8	0.0607
−9	0.0701

being done to actively reduce or maintain the size of the project (e.g. due to refactoring). Upon investigation of the 7 projects that decrease in size, no particular pattern could be identified in terms of programming language or otherwise. Overall, the evidence supports the rejection of H_20, providing support for law two.

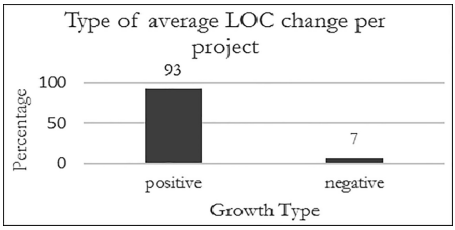

Fig. 3. Number of projects where LOC increased or decreased over time

4.3 H_3a – Issues, Additions and Deletions Over Time for Are Normally Distributed

To capture the essence of the third law three metrics would have to be considered to represent the 'products and process measures' and the 'self-regulating' keywords, in this case additions/deletions in tandem with issues was chosen. In order to determine if these measures were close to normal the Shapiro-Wilk test of normality was leveraged for each metric extracted from the 100 sampled projects. Again the 100 calculated p-values were compared with an alpha of 0.05.

Table 4 shows the overall results showing clearly that we cannot reject the H_30 hypothesis. Rather, evidence points to the non-normality of the data and counters law three despite threshold relaxations from the traditional 0.05. We also conducted an Anderson-Darling statistical test for normality and found a similar result. The

Table 4. Percentage of sampled distributions with varying alpha values for normality using Shapiro-Wilk

Alpha	Percentage of issues	Percentage of deletions	Percentage of additions
0.01	88%	100%	100%
0.02	90%	100%	100%
0.03	90%	100%	100%
0.04	91%	100%	100%
0.05	94%	100%	100%

non-normality may reflect the nature of open source development in which changes to the master branch can be made dynamically at any time, as a consequence of this it is possible that there will be periods where no change to the code is made. As a result of this the amount of additions and deletions may fluctuate from week to week with no consistency in the amount of code change, depending on the nature of the change which could vary from a minor bug fix to integrating a new feature. The OSS paradigm thrives upon contribution from distributed collaborators at any point in time, pull requests are monitored by the core projects team but a change is reviewed and accepted at any arbitrary point in time which disrupts the normality of additions and deletions.

4.4 H_4a – Project Issues Increase as Code Churn Decreases

We have limited the scope of our study to what can be extracted via the GitHub API. Further studies are ongoing looking at separate metrics for source code quality and its relationship to software change as determined from GitHub. Nonetheless, for completeness we have made the link from quality to the number of issues that occur in each weekly interval. LOC change will be used to assess code churn. To determine if a decrease or stagnation in the lines of code will lead to an increased number of issues (or vice versa) in the set of projects a cross correlation was again applied with various lag parameters tested to supplement the analysis. The main target was to evaluate each generated correlation value and count the amount of times for each of the one hundred projects that a negative correlation occurs, this has been expressed as a series of percentages in Table 5.

A pattern can be observed which shows the overall percentage increasing as the LOC lag is moved further into the past. This indicates that an increase/decrease in the LOC of a project will have a greater impact on the amount of issues over an extended period of time rather than immediately. Logically this makes sense as introducing new features in the past may typically spawn issues that were not immediately evident to the core team and may quickly to come to the surface following extensive usage and feedback from the user base. This would explain why the amount of positive correlations decrease as the lag is increased as a new feature may be introduced that has a side effect that produces bugs whereas if the −9 lag is considered the amount of issues will have decreased as the potential problems will have already been fixed by the present point in time from that initial change to lines of code.

Table 5. Percentage of correlations for different lags that are negative

Lag amount	Percentage of negative correlations **
0	32%
−1	33%
−2	32%
−3	32%
−4	35%
−5	35%
−6	37%
−7	37%
−8	36%
−9	38%

If the percentages themselves are considered it indicates that the amount of negative correlations in this context is the minority result, rather than changes in lines of code decreasing the amount of issues in most cases the amount of issues increase (or rather than stagnation/decrease in LOC introducing more issues, it reduces the amount of issues). This brings up a possible facet of open source development that may contribute to this phenomenon, typically a subset of the core team reviews pull requests and decides on whether to merge them or not. This potentially isolates a sizeable proportion of the contributors who have no input on what is accepted. Therefore, it is likely that in hindsight after the pull request was accepted and it has been extensively utilised that issues arise after an arbitrary amount of time. Table 5 suggests that considering the LOC further into the past has a lesser impact towards issues than immediate changes. Overall this suggests that an increase to lines of code is more likely to spawn an issue than stagnation or a decrease. This could be a result of introducing new features which could have only been tested in isolation by the core team and when exposed to the public give rise to more issues. On the other hand, a reduction in LOC could be a result of refactoring or removing dead code, thereby improving the software and preventing future problems.

To conclude reference should be made back to the hypothesis to determine an outcome, the results in Table 5 provide evidence which disputes law seven in most cases. Figure 4 shows a distribution of values that appear random indicating little support for H_4. Further, this claim which albeit contrast with the significant negative presence as seen in Table 5 shows only a minor affinity towards a positive correlation in the case of a mean.

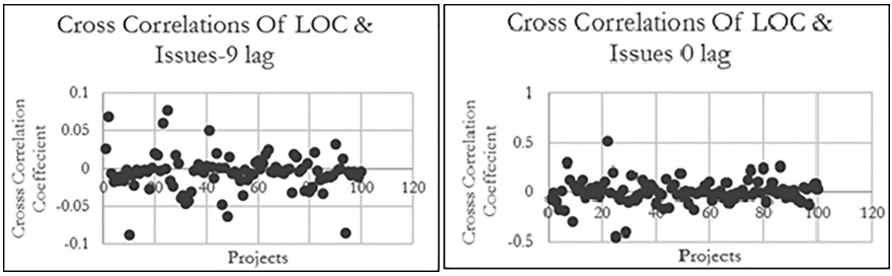

Fig. 4. Distribution of correlations at different lag points

5 Validity Discussion

5.1 Construct Validity

Due to a focus on the metrics that can be attained from the GitHub API, Lehman's laws had to be interpreted into hypotheses that represent the intent of each law as accurately as possible. In some cases logical metrics were available such as using stargazers to measure 'satisfaction', however in other cases there is room for dispute. An example of this is evidenced in law two 'increasing complexity' this study represents complexity as lines of code, however it is also possible to choose more appropriate measures such as McCabe's cyclomatic complexity which would involve delving into lower level metrics at the code base, which is beyond the scope of this initial study. Hypothesis six focuses on quality, the metric that has been attached to this law is issues and its relationship with code churn (additions and deletions) but in reality this is a much coarser grain aspect. A more complete study could account for testing code coverage, architecture, count of bugs among others.

In most cases the percentage values yielded at each lag interval indicate that a negative correlation is the majority result supporting the idea that the number of issues is to an extent driven by the volume of comments. Therefore the more a project team utilises a feedback system the more likely that the code will improve. It is difficult to explain why that in most cases a significant subset of the projects do not adhere to this principle, the size and make up of each team a transient factor that cannot be quantified in this context. In addition to this it is possible that as interaction via comments increases that this will lead to the discovery of additional issues that are associated with the current point of discussion.

The evaluation process for each hypothesis should also be taken into account, for H_1 and H_4 a binary threshold was used to generate the percentages at each lag interval. This does not account for the strength of each individual correlation value and how significant it may be, for example based on upon the scatter graphs provided in each of those hypotheses a broad subset of the data in most cases is focused around the zero point and may often times is extremely close to either being positive or negative. This lack of precision, while useful for stimulating a discussion may represent values that do no lean either way to supporting or refuting the hypotheses as significance is not easy to evaluate.

The pre-processing of the dataset also has the potential to impact the validity of the results, the first six months of each data point being trimmed from the evaluation to account for projects migrating to GitHub and the initial dump of data associated with that process. This process is indiscriminate of the whether a migration has occurred or not, so projects who have spent their entire life span on GitHub will also be targeted, this directly removes the possibility of analysing the early stages of evolution for these particular projects.

It should also be noted that the rate of activity on each project has not been a deciding factor in the selection process. Therefore, it is possible that among the range of projects there will be some that are maintained much more effectively than others, this is dependent on factors such as the size of the team actively working on the project and the amount of general user collaboration on GitHub. This might lead to cases where the activity of the team itself becomes a driver of software evolution, which this study does not account for and could be an avenue for future work.

5.2 External Validity

While the random selection of a fairly large set of projects was made the study would benefit from repetition before any attempt is made to generalise the findings. The study targeted the most popular projects on GitHub as representative of typical open source development. This is open to dispute and the findings may not necessarily apply to all projects e.g. less popular projects. We also targeted popularity as defined by developer attention rather than attention from users.

5.3 Conclusion Validity

Using the results, in most cases it is difficult to unequivocally support or refute a law. The main contributions of the paper are therefore in the discussion of the results and support or otherwise for Lehman's laws. In some cases, (e.g. H_2 and H_3) the results provide clear support or refutation However the other hypotheses remain much more debateable.

6 Conclusions

The goal of this paper was to evaluate Lehman's Laws of software evolution via usage of the GitHub API. Our research question on whether it is possible using data extracted from the GitHub API to determine if OS software evolution over time reflects Lehman's laws is answered. There is much that can be extracted to help study software evolution and to evaluate Lehman's laws and tools can be built that make the process of data extraction, transformation and analysis relatively straightforward. The experience shows that the data that can be extracted from GitHub is insufficient to conclusively test Lehman's laws. A much richer dataset, perhaps including data on downloads, on defects or on quality would produce much more useful results.

On support or otherwise for Lehman's laws we found contradictory evidence in all cases studied except for laws 2 and 3. For H_2 i.e. that the total lines of code increases as

a software system evolves we can reject the null hypothesis and therefore find support for law two. On the other hand, we found evidence to support H_30. Here the evidence seems to clearly reject law three since we found in our study that issues, additions and deletions over time were not normally distributed.

Our findings are often similar to those in related work as discussed in Sect. 2. The work by Gonzalez-Barahona [7] also found support for law two and also questioned the validity of law three. Similar to our work they have found the evidence for the other laws to be inconclusive and, interestingly, question the falsifiability of the laws, especially given the need to use proxy measures such as LOC for complexity. The work by Mens et al. [18] and Fernandez-Ramil [19] on Eclipse found support for laws one and six, but otherwise found their data inconclusive. They did find partial support for law two depending on the complexity metric used. The message here may be that support or otherwise for Lehman's law depend on the indicators used. As an example there is no agreement on how to measure such things as complexity or satisfaction. Even more so the best measures are not always practicable and proxy measures must be used.

Overall, in our findings, based on results from a dataset of 100 open source projects, only one of the hypotheses provide enough evidence to support the laws investigated. The discussion for why this occurs often reflects the context of open source development and the GitHub platform itself which are aspects of software evolution that Lehman's laws did not foresee. However, utilising only data that can be extracted from the API at the repository level imposed certain restrictions on the nature of each hypotheses interpretation therefore further work into this topic could be explored that integrates a detailed analysis of the code base itself in order to supplement these findings. We are also looking to extend work on causes of change [24, 25] as it applies to OSS. In addition to this future contribution may entail presenting an alternative to Lehman's laws which fully consider the open source paradigm and establish a set of rules that account for the variations in this approach from traditional software development.

References

1. Paulson, J.W., Succi, G., Eberlein, A.: An empirical study of open-source and closed-source software products. IEEE Softw. **30**, 246–256 (2004)
2. Greer, D., Conradi, R.: Software project initiation and planning - an empirical study. IET Softw. **3**(5), 356–368 (2009)
3. Lehman, M.M.: On understanding laws, evolution, and conservation in the large-program life cycle. J. Syst. Softw. **1**, 213–221 (1980)
4. Lehman, M.M.: Laws of software evolution revisited. In: Montangero, C. (ed.) EWSPT 1996. LNCS, vol. 1149, pp. 108–124. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0017737
5. Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D.M., Damian, D.: The promises and perils of mining github. In: Proceedings of 11th Working Conference on Mining Software Repositories, pp. 92–101 (2014)
6. GitHub Features (2019). https://github.com/features. Accessed 31 Mar 2019

7. Gonzalez-Barahona, J.M., Robles, G., Herraiz, I., Ortega, F.: Studying the laws of software evolution in a long-lived FLOSS project. Softw. Evol. Process **26**(7), 589–612 (2014)
8. Syeed, M., Hammouda, I., Syatä, T.: Evolution of open source software projects: a systematic literature review. J. Softw. **8**(11), 2815–2829 (2013)
9. Yu, L., Mishra, A.: An empirical study of Lehman's law on software quality evolution. Int. J. Softw. Inform. **7**(3), 469–481 (2013)
10. Sheoran, J., Blincoe, K., Kalliamvakou, E., Damian, D., Ell, J.: Understanding 'watchers' on GitHub. In: Proceedings of 11th Working Conference on Mining Software Repositories, pp. 336–339 (2014)
11. Ben, X., Beijun, S., Weicheng, Y.: Mining developer contribution in open source software using visualization techniques. In: Proceedings of 3rd International Conference Intelligent System Design and Engineering Applications (ISDEA), pp. 934–937 (2013)
12. Gousios, G., Pinzger, M., van Deursen, A.: An exploratory study of the pull-based software development model. In: Proceedings of the 36th International Conference on Software Engineering, pp. 345–355 (2014)
13. Borges, H., Valente, M.T., Hora, A., Coelho, J.: On the Popularity of GitHub Applications: A Preliminary Note. arXiv:1507.00604 (2015). Accessed 31 Mar 2019
14. Bissyandé, T.F.: Got issues? Who cares about it? An investigation of issue trackers of 105 projects. In: Proceedings of IEEE 24th International Symposium on Software Reliability Engineering, pp. 188–197 (2013)
15. Neamtiu, I., Xie, G., Chen, J.: Towards a better understanding of software evolution: an empirical study on open-source software. J. Softw. Evol. Process **25**, 193–218 (2013)
16. Skoulis, I., Vassiliadis, P., Zarras, A.: Open-source databases: within, outside, or beyond Lehman's laws of software evolution? In: Jarke, M., et al. (eds.) CAiSE 2014. LNCS, vol. 8484, pp. 379–393. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07881-6_26
17. Israeli, A., Feitelson, D.G.: The Linux kernel as a case study in software evolution. J. Syst. Softw. **83**(3), 485–501 (2010)
18. Mens, T., Fernández-Ramil, J.: The evolution of Eclipse. In: Proceedings of International Conference Software Maintenance, pp. 386–395 (2008)
19. Fernandez-Ramil, J., Lozano, A., Wermelinger, M., Capiluppi, A.: Empirical studies of open source evolution. In: Mens, T., Demeyer, S. (eds.) Software Evolution, pp. 263–288. Springer, Berlin (2008). https://doi.org/10.1007/978-3-540-76440-3_11
20. Stroulia, E., Kapoor, R.: Metrics of refactoring-based development: an experience report. In: Wang, X., Johnston, R., Patel, S. (eds.) OOIS, pp. 113–122. Springer, Heidelberg (2001). https://doi.org/10.1007/978-1-4471-0719-4_13
21. Graylin, J., Hale, J., Smith, R.K., Hale, D., Kraft, N.A., Ward, C.: Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship. J. Softw. Eng. Appl. **2** (3), 137–143 (2009)
22. La, A.: Language Trends on GitHub. https://github.com/blog/2047-language-trends-on-github. Accessed 31 Mar 2019
23. RFoundation: The R Project for Statistical Computing. https://www.r-project.org/. Accessed 31 Mar 2019
24. McGee, S., Greer, D.: A software requirements change source taxonomy. In: 4th International Conference on Software Engineering Advances, ICSEA 2009, Includes SEDES 2009: Simposio para Estudantes de Doutoramento em Engenharia de Software (2009)
25. McGee, S., Greer, D.: Towards an understanding of the causes and effects of software requirements change: two case studies. Requir. Eng. **17**(2), 133–155 (2012)