Do Lehman’s Laws of Software Evolution apply to Open Source Software? A study using GitHub API.

A dissertation submitted in partial fulfilment of

The requirements for the degree of

MASTER OF ENGINEERING in Computer Science

In

The Queen's University of Belfast

By

Jordan McDonald

**\*ADD SUBMISSION DATE HERE\***

**\*** **Declaration Cover Sheet ADD\***

**ACKNOWLEDGEMENTS**

I would like to take this opportunity to show my appreciation to Dr Desmond (Des) Greer for his time, effort and diligence in assisting me in making this project a success. Not only did I receive excellent advice and guidance but he opened up the possibility of sharing my work via publication which will prove to be the biggest pat on the back I can imagine. It is no understatement to say that without his intervention the potential for this research and development project would not have been realised – Thank you Des!

**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 have changed very significantly 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, while 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 each of Lehman’s laws is evaluated using the data available. The study has discovered that the evidence does not support many of the laws, but with the proviso that the study is limited by 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**

The term software evolution represents the change of a software system as time progresses, factors that instigate this change include various forms of maintenance which can be categorised as adaptive, perfective, corrective and preventative[15]. To evaluate this change Lehman and Belady formulated the laws of software evolution, which attempted to outline the factors that drive growth and development of software, while also taking into account forces that lead reduced progress. Lehman theorised that most software is subject to change over the course of its existence and this change can be driven by a multitude of different events. The goal was to identify a set of laws that these changes would obey, or must obey in order for software to survive (Section II.A).

The goal of this paper is to examine these laws in the context of modern open source projects hosted on GitHub, with a dataset mined from the GitHub API as the focal point for the study. GitHub is a hosting service designed for collaboration on a centralised 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 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. Users also have profiles that can be populated with identifying information and encapsulate their recent activity within the site [2].

At the time of writing GitHub reports having 14 million users and 35 million repositories [3] making it the largest host of source code in the world [4]. This represents a period of rapid growth considering in 2010, announced on the official GitHub blog it was revealed that one million repositories were hosted on GitHub. These factors in tandem with the accessible GitHub API’s data on commits, code churn, issues, watchers and pulls among other metrics provide an excellent foundation to examine Lehman’s laws to determine if they apply in the context of 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 if Lehman’s laws hold or are contradicted by the findings. Each law will be represented by metrics taken from the API and the evolution of these metrics over time will provide an insight into software growth patterns, which in turn shall test the validity of the laws devised by Lehman. Section III will describe the background to the work, Section IV will describe the methodological aspects of the study, and Section V will discuss the results of the work while section VI will identify threats to validity in the study. Finally, in Section VII we attempt to draw conclusions from the evidence and discuss avenues for future research.

1. **BACKGROUND AND RELATED WORK**

**A. Background**

Initially devised in 1974 Lehman’s laws have undergone multiple changes as the years have progressed, with the latest alteration taking place in 1996. In his 1980 article [5] Lehman qualified the application of such laws by distinguishing between three categories of software – S-programs (exact specification of what that program can do), P-programs (implement certain procedures that completely determine what the program can do) and E-Programs (perform some real-world activity, linked to the environment it runs on). It is evident that the laws reflect the E-program definition devised by Lehman, the emphasis on feedback and adaptations of software are key components of evolution. Each project in this study will in turn reside under the E-program umbrella and each law is applicable to this category, see figure 1 for a summary of each.

|  |  |
| --- | --- |
| **Law #** | **Law Description** |
| 1 | (1974) **"Continuing Change"** - an E-type system must be continually adapted or it becomes progressively less satisfactory |
| 2 | (1974) **"Increasing Complexity"** - as an E-type system evolves, its complexity increases unless work is done to maintain or reduce it |
| 3 | (1974) **"Self-Regulation"** - E-type system evolution processes are self-regulating with the distribution of product and process measures close to normal |
| 4 | (1978) **"Conservation of Organisational Stability (invariant work rate)"** - the average effective global activity rate in an evolving E-type system is invariant over the product's lifetime |
| 5 | (1978) **"Conservation of Familiarity"** - as an E-type system evolves, all associated with it, developers, sales personnel and users, for example, must maintain mastery of its content and behaviour to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves. |
| 6 | (1991) **"Continuing Growth"** - the functional content of an E-type system must be continually increased to maintain user satisfaction over its lifetime |
| 7 | (1996) **"Declining Quality"** - the quality of an E-type system will appear to be declining unless it is rigorously maintained and adapted to operational environment changes |
| 8 | (1996) **"Feedback System"** (first stated 1974, formalised as law 1996) - E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base |

Figure 1 – A table that states each of Lehman’s laws of software evolution

**B. Related Work**

Attempts at general data mining from GitHub has been prominent in recent years, Kalliamvakou et al [2] published a paper that highlighted the ‘promises and perils of mining GitHub’. This paper has a focus on avoiding common pitfalls in GitHub mining and concluded that there is valuable data to be found if these are avoided. M.M. Mahbubul Syeed [11] has previously performed a systematic literature review into the evolution of open source projects, the authors examine the data sets utilised, sources of the data and research trends in recent years. The author found that Lehman’s laws do not hold in certain cases, with individual laws in the research yielding contradicting results in regards to open source projects.

Additional papers have provided much more focused studies, Jyoti Sheoran et al [7] investigate the watcher mechanic on GitHub, which provides notifications to user who watch a repository each time an event occurs such as a commit or creation of an issue. The paper hones in on the contributors of a project, tracking to 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 Xu Ben et al [9] which performed visualisation on metric related to commits, low level code statistics and lines of code on a single project, this restriction limits the usefulness of the research. Georgios Gousios et al [4] 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 in previous literature, 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. [11] Provides insight into what constitutes a projects popularity on GitHub using the starring mechanic, the paper theorised that this could be tracked over time to show the evolution of popularity. [13] Analyses issues (bugs) as part of open source software, correlating the data with watchers, forks and other metrics.

A similar study to that presented in this paper in regards to evolution was performed by Jesus M. Gonzalez-Barahona et al [8] was conducted on a long running FLOSS project, glibc inside a SCM repository with over 20 years of history. The paper also approaches the research through reference to Lehman’s laws. The metric utilised has a focus on commits, lines of code and files changed to represent evolution – a downside to this study is single project focus, this paper hopes to consider a much larger dataset in order to draw novel findings. [17] [18] take a single and seven project approach respectively with a focus on long running projects such as SQLite and the open source browser Firefox. [19] Has a sample size of nine projects and utilises code level metrics such as KLOC [10] also delves into software evolution and Lehman’s law, however from the context of databases. [21] Focuses on 810 versions of a single project, the Linux kernel utilising a process that quantifies each law using metrics such as LOC and McCabe’s cyclomatic complexity applied to the source code. This study found that several of Lehman’s laws were supported by the data and made novel observations regarding software evolution in the case of this particular project.

It also important to consider research that focuses on the laws outside of the given OSS context in this paper [20] provides an exploratory discussion on the validity of the laws on component based software with the intent of enabling future work on the topic. [22] Briefly investigates the laws in the context of a single project case study utilising refactoring based development and concludes that the laws apply utilising this approach. Attempts have been made to applying Lehman’s laws to instantiating new development models [23] proposing the ‘staged model’ which focuses on the truism of each law and how each ‘stage’ reaffirms Lehman’s laws.

**C. Novel approaches in this paper**

On conclusion of the literature review gaps in the research were identified from which novel contributions to the field could be made. Evaluating Lehman’s laws according to data from the GitHub API has not, to the author’s knowledge, yet been fully investigated. This paper plans to represent each law with a relevant metric(s) and quantify the evolution of these data points over a sustained period of time. Prior studies that are similar to the approach in this paper have flaws – A) only investigating one project B) looking at evolution from the stand point of databases. This study will encompass a large data set with variation in the language of choice for the repositories, from this it will be possible to determine if a pool of different programing languages will support or contradict Lehman’s laws.

1. **PROPOSED METHODOLOGY**

**A. Research Questions & Hypotheses**

In order to provide scope and direction to the research we have defined a research question with multiple hypotheses that will attempt to draw out appropriate relationships for each law with the metrics extracted via the GitHub API. Each (null) hypothesis is also accompanied by justification and reasoning which explain the choice of metric used to represent each law.

**RQ1** - Is it possible using data extracted from the GitHub API to determine if OS software evolution over time reflects Lehman’s laws?

**H1** – If the amount of commits decreases the amount of star gazers will also reduce (law 1 & 6)

**H10** – 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 exist as a result of user demand and it is often the case that the user of an open source project is also a developer so by proxy if a developer stops stargazing we can assume that the software has become less satisfactory and therefore represents the law sufficiently. Commits has been used to represent continual adaptation which encapsulates all changes made to the code base for a specific purpose (adding a feature, refactoring etc.) and therefore proves a suitable metric to measure this aspect of the law.

**H2** – Total lines of code increases as software system evolves (law 2)

**H20** – Total lines of code will not increase as a software system evolves

**Justification** – To represent complexity LOC has been selected as the most suitable measure due to the scope of this study focusing on API attainable metrics over source code analysis. There is good support in literature [24] that suggests that LOC and McCabe’s cyclomatic complexity have a stable linear relationship which provides support for this choice, extending to utilise other complexity measures remains in plan for future work.

**H3** – Issues, additions and deletions over time for will be normally distributed (law 3)

**H30** – Issues, additions and deletions over time for are not normally distributed

**Justification** – To capture the essence of ‘self-regulation’ in particular issues was chosen as the metric which 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 reflected by the code output, at a lower level additions and deletions which is a more appropriate choice than commits as it accounts for both addition of new features (and as a side effect increased additions) and maintenance/refinement (refactoring) which provides a finer grain analysis of the source code.

**H4** – As software evolves changes to lines of code should remain invariant over time (law 4)

**H40** – As software evolves changes the quantity of changes to lines of code will not remain invariant over time

**Justification** – in this case the ‘global activity rate’ is represented by LOC which reflects the entirety of the project and its contents, therefore it sufficiently meets the ‘global’ requirement as all modifications to the source code will impact the LOC. In addition to this if the phrase ‘work rate’ is considered, this in a software context relates closely to changes to the code base and as a consequence of this the LOC which reinforces the selection of this measure.

**H5** – As Lines of code increases the amount of issues will also increase (law 5)

**H50** – There is no relationship between Lines of code and the number of issues.

**Justification** – To represent an invariant growth of the project the lines of code is an ideal choice however an assumption is made that negative growth (reduction in LOC) will have a positive impact on the project. It is possible that a significant decrease in LOC as a result of refactoring may introduce more issues and reduce mastery but to capture the essence of the law an increase in growth is the focal point of this hypothesis. Issues has been chosen to account for the ‘mastery’ requirement, if a developer has reduced familiarity as a result of significant growth it may lead to increased issues/bugs in the future due to misinterpreting new behaviour or functionality. Issues has been chosen over stargazers to represent ‘satisfactory evolution’ as significant growth that exceeds the invariant could be interpreted by passive stargazers as a thriving project which is well supported and thus will continue to stargaze and may even lead to an increase in stargazers who note this activity.

**H6** - Project issues will increase as code churn decreases (law 7)

**H60** – 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 and more issues will be created by the developers to account for this reduction. Code churn will be represented by lines of code which satisfies the need for rigorous maintenance to prevent a decline in quality and this is reflected in the hypothesis.

**H7** – As the number of issue comments increases the number of issues should decrease (law 8)

**H70** – There is no relationship between the number of issue comments and the number of issues.

**Justification** – As the main form of feedback in the GitHub environment issues was an ideal measure, and to capture the actual feedback from developer’s comments on this issues would be crucial to evaluating the volume of feedback. In addition to this issues can be used to measure a degree of improvement, an assumption is made that a reduction in issues leads to improved code, from this the hypothesis can determine how the feedback system increases/decreases the amount of issues.

**B. Project Selection**

To provide scope to the research performed in this paper, a process of identifying the volume and variation of the projects attained from GitHub needs to be defined - figure 2 demonstrates the selection process. The ten programming languages of choice have been chosen based on a ranking system seen in the GitHub blog post [14] which shows the top ten used languages (based on total active repositories) on the site in public and private repositories (excluding forks) as of August 2015.

Top 10 languages on GitHub [14]

1. JavaScript
2. Java
3. Ruby
4. PHP
5. Python
6. CSS
7. C++
8. C#
9. C
10. HTML

Identify the top ten languages

Search each language for the most popular project (sorted by stars)

Has the current project been on GitHub for five years or more?

Does the project have a fifty percent plus affinity to the target language & metrics fully populated?

Add the project to the selection for analysis

If total projects for the current language equals ten?

Select another programming language

Figure 2 - Flow chart showing the project selection progress

It is crucial to apply restrictions to the projects selected for each programming language in order to visualise the evolution of the software effectively and maintain the integrity of the target programming language requirement. The GitHub advanced search facility on the site allows the descending ordering of the ‘most stars’ for a programming language, each sequential project is then evaluated against two criteria.

1. Duration of project life on GitHub, with a set five year threshold which is chosen to ensure evolution can be mapped over a sustained period of time.
2. It is very common for most projects to use multiple programming languages, however GitHub allows users to examine a project for the breakdown of languages utilised. Using this each project prior to analysis has to meet the 50% target language affinity requirement.

This process will be applied to two hundred projects in total, the final dataset of one hundred will then be randomly selected with the intent of taking ten projects from each programming languages group of twenty.

**C. 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 three and all requests are performed over HTTPS, the data is returned in a JSON format which allows simplistic parsing of the metric required. Disadvantages to the API include the pagination system which restricts the amount of data that can returned in one request, which may lead to multiple similar requests taking place (with an alteration in target page number).

**D. System Design**

To enable the research a workbench (figure 3) has been devised which will handle the automated collection of the data for each of the two hundred projects and to execute the statistical functions. To interact with the GitHub API the JQuery library will be leveraged in order the extract the relevant data via a HTTP call using Ajax, the response from the API will take the form of JSON which can then be parsed as required. In order to answer the hypotheses various statistical methods would need to be applied, to handle this the R environment was integrated in the workbench in order to reliably get results using the built in libraries of R.

**D.1 Overall System Architecture**

GitHub API

Webpage

URL(s)

JSON data

Java Servlet

JSON extractor module (JS)

Raw data

MongoDB

DB Query

R Environment

Figure 3 – GitHub extraction workbench architecture

**E. Data Analysis Methods**

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 to account for projects that have origins that outlive the GitHub platform. The reason for this is to remove 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, the relationship between these and the hypotheses has been covered in a previous section (with additional metrics added for the flexibility of the workbench).

* **Stargazers** - Repository Starring is a feature that lets users bookmark repositories. Stars are shown next to repositories to show an approximate level of interest [16].
* **Commits** - A commit, or "revision", is an individual change to a file (or set of files).
* **Additions & Deletions** – represent modified, added or removed lines of code.
* **Issues** - Issues are suggested improvements, tasks or questions related to the repository.
* **Issue Comments** – Messages that a user has attached to a specific issue.
* **LOC** – total lines of code at a certain point
* **Growth Rate** – how much a certain metric changed per time interval

**E.1.**  **Statistical Methods**

**Growth Rate**

This equation has significant value in the context of software evolution, where values are analysed over a sustained period of time. In particular working in tandem with an LOC metric to provide evidence for hypothesis two highlighting an ideal use case for this statistic. In addition to this it will have value when applied to hypothesis five, but in this case it will be mapped to a time series and cross correlated with issues.

1. Growth Rate -

2. Average Growth Rate –

X = current value

Y = past value

N = total samples

**Shapiro Wilks Test**

This particular test will be applied to the three metrics stated in hypothesis three in order to determine the distribution of the data and evaluate the normality. This particular statistic utilises the null hypothesis principle (The null-hypothesis of this test is that the population is normally distributed) using a set alpha (0.05 in this case) if the p value is below this threshold then the null hypothesis is rejected and there is evidence that the data tested is not from a normally distributed population.

**Cross Correlation**

To adequately answer hypotheses one, five, six and seven 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.

**Variance & Standard Deviation**

Law four concerns itself with an invariant work rate, this can be interpreted to applying a variance on the growth rate of the projects LOC. The growth rate will become a series of growth rate values between each weekly LOC, the variance will be applied to this series and a medium operation will be applied to the priori generated one hundred variances in order to facilitate discussion while standard deviation will be utilised to measure the ‘distance’ of the elements within a growth rate metric.

**V. RESULTS**

**A. – H1: If the amount of commits decreases the amount 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 stargazers and in particular what duration is of time after a commit is the change felt most significantly. The results of this experiment are shown in figure four which shows the results with a lag ranging from -9 to 0 weeks.

The results presented in figure four 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.

|  |  |
| --- | --- |
| Lag Amount | Percentage of 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% |

Figure 4 – percentage positive cross correlation at different lag intervals between #commits and #stargazers

The next step will be to consider the significance of the percentage value towards accepting or discarding the null hypothesis. The value itself for all lags is not conclusive enough to be able to determine this, however 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 which 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. As supporting evidence for this hypothesis figure six has been provided which visualises the distribution of correlation values at different lag intervals, highlighting the almost random nature of the values obtained are further strengthening the null hypothesis. In addition to this figure five shows the mean correlation values for each lag interval and while all remain positive the significance of this is extremely minor reinforcing the plots in figure five which show no clear pattern in the distribution of the data.

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.

|  |  |
| --- | --- |
| Lag Interval | Mean Cross Correlation for #commits and #stargazers |
| 0 | 0.03214124591164291 |
| -1 | 0.025790287943452533 |
| -2 | 0.026634282854046863 |
| -3 | 0.030738468993725624 |
| -4 | 0.03359798856683962 |
| -5 | 0.03805715117538213 |
| -6 | 0.04605641540345248 |
| -7 | 0.03670980352004324 |
| -8 | 0.06072350694356765 |
| -9 | 0.07010929633573694 |

Figure 5 - shows the mean cross correlation for stargazers and commits organised by interval

Figure 6 – graphs showing the distribution of correlation values for each of the 100 projects on different lags intervals

**B - H2: Total lines of code increases as 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 seven visualises the results of this process, the majority of the projects do increase in size as the software system evolves. This is generally to be expected as time progresses the demand for new features and functionality to improve on the existing software will be constant in order to maintain a user base, this is particularly crucial in open source software where new libraries and technologies are introduced at a rapid frequency. However there remains several projects that have confounded the hypothesis and reduced in size, law two states that this could be the side effect of work being done to actively reduce or maintain the size of the project. Reasons that this could occur is refactoring, which is a prominent part of software evolution and certain projects may have taken steps to streamline or alter the architecture of the system. Upon investigation of the seven projects that decrease in size no particular pattern could be identified in terms of programming language or other factors so an assumption could be made that the reasons discussed prior could account for this. Overall, the evidence supports the rejection of H20, providing support for law two.

Figure seven – shows the amount of projects whose LOC increased or decreased over time

**C - H3: Issues, additions and deletions over time for will be 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-Wilks test of normality was leveraged for each metric extracted from the one hundred projects. In order to determine the significance of the measure the p-value was utilised which could then be compared to an alpha (0.05) to determine if the null hypothesis (the population is normally distributed) could be accepted, from this percentages could be generated showing the amount that are within the threshold.

Figure eight shows the overall results of this process which indicates that the null hypothesis cannot be rejected, rather, evidence points to the non-normality of data which counters laws three despite the threshold relaxation. An Anderson Darling test of normality was also performed (figure 9) and yielded extremely similar results. This reflects 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. Once issues are observed the result is not as conclusive, this could be down to factors such as

a) The volume of issues per week typically is much lower than additions and deletions which would reduce the scope for the same extremes of fluctuation.

b) Issues in GitHub terminology could also be opportunities to refactor/improve the code base and as time progresses and more features are added to the software it is likely that issues would continually be identified by the users or development team.

However even when these points are considered the majority of the projects issues are not normally distributed which would indicate that the hypothesis and the law itself are refuted due to the overwhelming evidence provided. The reason for this once again is a product of the open source paradigm which 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 particularly additions and deletions, lending support to the null hypothesis

|  |  |  |  |
| --- | --- | --- | --- |
| **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% |

Figure 8 – percentage of distributions with varying alpha values using Shapiro Wilks

|  |  |  |  |
| --- | --- | --- | --- |
| **Alpha** | **Percentage of Issues** | **Percentage of Deletions** | **Percentage of Additions** |
| 0.01 | 88% | 100% | 100% |
| 0.02 | 88% | 100% | 100% |
| 0.03 | 90% | 100% | 100% |
| 0.04 | 93% | 100% | 100% |
| 0.05 | 94% | 100% | 100% |

Figure 9 - percentage of distributions with varying alpha values using Anderson Darling

**D - H4: As software evolves changes to lines of code should remain invariant over time**

In order to determine an invariant work rate LOC was chosen as the measure which was then applied to the growth rate algorithm which measured the amount of weekly growth at each point of the projects life span in order to generate a vector of percentage growth rate values. The variance of each vector was then extracted in order to determine how much of an instability in ‘work rate’ was present in each project, see figure ten to view the distribution of variance for each of the one hundred projects. From these graphs it is clear that the growth rate variance for each project can fluctuate between different extremes, the highest values are representative of projects whose growth is unpredictable, possibly due to sudden significant shifts in growth or may have long periods with no change to growth rate that precede an a spike in contributions. It is possible to observe significant outliers that are prominent in the set of variance values, therefore to aid in interpretation the median of these values was calculated – 30.290. This needs further analysis and investigation as it may reflect the practice of sudden dumping of code into a repository in some projects

As a result of the presence of outliers it is difficult to determine an outcome to the null hypothesis, to represent a reasonable invariant growth rate the standard deviation for each projects growth rate vector was calculated. Since the amount of lines of code that change per weekly interval may vary based on a number of factors, introducing the standard deviation as to act as a threshold to determine a reasonable distance from the mean would prove useful (figure eleven). The vast majority of each projects vectors are showing significant affinity to the one standard deviation invariant work rate threshold which suggests that the over the course of the projects life cycle the lines of code changes remain within a reasonable level of invariance. However this does not account for the growth rate values outside of the threshold which may represents growth that is among the more extreme cases, however it is reasonable to assume that over the course of a systems life span there will be changes that are more significant than the norm. Overall, the test for H4 is rather inconclusive and offers little support for Lehman’s fourth law.

Figure 10 – distribution of LOC growth rate variance for each of the one hundred projects

Figure 11 – % of each projects growth rate values within one standard deviation

**E - H5: As Lines of code increases the amount of issues will also increase**

Law five "Conservation of Familiarity" suggests that excessive growth off software as time progresses will reduce the mastery of the user base and lead to reduced satisfaction. To measure this LOC has been chosen to represent growth and issues has been utilised as an indicator of user bases mastery of the software which can then be applied to a cross correlation to determine if a positive correlation is present. Figure 12 shows the results of this process by calculating the percentage of cross correlation values for each project at different lag points that show a positive correlation.

|  |  |
| --- | --- |
| Lag Amount | Percentage of positive correlations \*\* |
| 0 | 50% |
| -1 | 50% |
| -2 | 49% |
| -3 | 46% |
| -4 | 51% |
| -5 | 44% |
| -6 | 41% |
| -7 | 44% |
| -8 | 50% |
| -9 | 45% |

Figure 12 - % of cross correlation values for each project showing a positive correlation

Initially a discussion will be made on the impact of applying a lag to the LOC growth rate has on its correlation with the amount of issues generated by users for the projects. Particularly the negative eight lag result which in comparison to those lags that represent the impact of a change in LOC in weeks closer to the ‘present’ time there appears to be no point that an increase/decrease in growth rate has an impact on the amount of issues. Reasons for this could include the sporadic nature of growth in open source projects which often do not confirm to a schedule, new code is often integrated on an ad-hoc basis and if a change introduces any problems (an issue) it may only become evident in a very specific use case at an arbitrary point in time before being reported. In addition to this Lehman focuses on ‘familiarity’ and ‘mastery’ which are not terms which cannot be applied to the dynamic potential pool contributors outside of an OSS project core development team who may develop the code base without in depth knowledge about the intricacies of the software.

The next step will be to interpret the percentage results that have been obtained at applying the cross correlation to varying growth rate lags. Overall the results do not show any relationship between the amount growth rate and the amount of issues and mostly highlights a random distribution of cause and effect in this case. Only one of the outcomes produce a majority positive correlation for each of the one hundred projects, based on this evidence the hypothesis will be rejected and in turn law five is refuted based on this dataset and interpretation. To reinforce this point a series of graphs has been presented in figure 14 which highlight the lack of relationship and random nature of the correlation values attained. In addition to this figure 13 presents the mean correlation for each lag interval, the means show only an extremely minor preference to positive or negative values which would support the rejection of the hypothesis.

|  |  |
| --- | --- |
| Lag Interval | Mean Cross Correlation for LOC growth rate and #Issues |
| 0 | -0.009252038848645337 |
| -1 | 0.0047858464040523825 |
| -2 | 0.0038369126668407246 |
| -3 | -0.013259219448416919 |
| -4 | -0.014405203443080793 |
| -5 | 0.007688806578580345 |
| -6 | 0.007967327862363583 |
| -7 | 0.0045222165234004045 |
| -8 | 0.0033959314081396736 |
| -9 | 0.006964737309696906 |

Figure 13 – mean correlation value at each lag interval

Figure 14 – distribution of cross correlations at different lags

**F - H6: Project issues will 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. 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 figure 15.

|  |  |
| --- | --- |
| 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% |

Figure 15 – percentage of correlations for different lags that are negative

A clear pattern is evident which shows the overall percentage increasing as the LOC lag is moved further into the past. This indicates that an increase/decrease off 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.

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 facet of open source development that contributes this phenomenon, typically a subset of the core team reviews pull requests and decides on whether to merge an alteration to the code base. This potentially isolates a large proportion of the contributors who have a more ad-hoc presence who could foresee a bug/future issue that could come as a result of accepting a certain merge. Therefore it is likely that in hindsight after the pull request was accepted and it has been extensively utilised by the user base issues could arise after an arbitrary amount of time, the table in figure 15 suggests that changing the LOC further into the past have 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 more issues arise. Whereas a reduction in LOC could be a result of refactoring or removing dead code, therefore improving the software and preventing future problems that may arise.

To conclude reference should be made back to the hypothesis to determine an outcome, figure 17 shows a distribution of values that appear random which indicates the behaviour as defined by Lehman for this law is not being adhered to therefore strengthening the null hypothesis. To supplement the analysis figure 16 has been provided showing the mean correlation value at differing lags which albeit contrast with the overall negative percentage seen in figure 15, only demonstrate a minor affinity towards a positive correlation.

|  |  |
| --- | --- |
| Lag Interval | Mean Cross Correlation for #issues and #LOC |
| 0 | 0.06167135617049322 |
| -1 | 0.07120205221347067 |
| -2 | 0.07307856963449424 |
| -3 | 0.0768328803933144 |
| -4 | 0.08474302646843988 |
| -5 | 0.09042736936398382 |
| -6 | 0.09493397747520561 |
| -7 | 0.09402995122438083 |
| -8 | 0.09598072509317271 |
| -9 | 0.10322327947972923 |

Figure 16 – mean cross correlation for issues and LOC at different lag intervals

Figure 17 – distribution of correlations at different lag points

**G - H7: As the number of issue comments increases the number of issues should decrease**

Law seven focuses on a ‘feedback system’ and its effect on improving the code base, the major feedback facility on GitHub can be seen in issue tracking which allows users to report bugs and discuss new features and thrives on interaction in the form of comments to enable discussion. Based on this issue comments have been selected to represent the feedback system and issues itself will represent code improvement with a decrease in issues seen as a key factor as the amount of comments increase. Once again a cross correlation has been chosen as the measure to evaluate this hypothesis, with the percentage of negative correlations at each tested lag interval used to determine if the amount of comments indeed reduce the amount of issues.

|  |  |
| --- | --- |
| Lag Amount | Percentage of negative correlations |
| 0 | 53% |
| -1 | 61% |
| -2 | 49% |
| -3 | 53% |
| -4 | 59% |
| -5 | 60% |
| -6 | 61% |
| -7 | 65% |
| -8 | 64% |
| -9 | 69% |

Figure 18 – percentage of correlations that are negative for different lag points

If the results in figure 18 are observed there lies relationship between the size of the lag interval and the percentage of projects that output a negative correlation for these two metrics. Initially the zero lag will be considered which measures the effect of comments on the count of issues that occur in the same week, reasons this value is low could be attributed to the fact that discussion is ongoing about a particular issue and therefore there is less of an opportunity for an assignee to tackle the problem. This is a side effect of open source projects coordinating distributed teams which a certain amount of delay is to be expected due to the medium of communicating via a comments system, reducing the scope for rapid solutions to an issue and this is generally reflected in the majority of the smaller lag intervals. In contrast the more the lag applied the more likely that the amount of comments will drive the volume of issues, potentially active engagement in discussing an issue and coming to a consensus on the best solution prevents the possibility of a related issue (due to an original solution which was not fully considered) appearing in the future due to a pooling of contributor knowledge.

In most cases the percentage values yielded at each lag interval indicate that a negative correlation is the majority results which suggests that the amount of issues to an extent is driven by the volume of comments. Therefore the more a project team utilises a feedback system the more likely that the code will improve which reflects the benefits of open source development which allows a user bases of varying expertise to pool together and discuss a problem in order to discover the best possible solution. 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, as open source development thrives on ad-hoc contributors the possibly of a new perspective offering an opinion that was not previously considered is very real.

In terms of evaluating H7 the initial evidence across the 100 projects suggests rejection of the null hypothesis. However this support for H7 is tempered when considering the variation when different lags are considered and the fact that the dispersion of the correlations for each of the one hundred projects appears random (see figure 20). To support this figure 19 has been provided which shows the mean correlation value at each lag for each of the one hundred projects, the observed random distribution appears to be supported based on this due to no significant affinity to either positive or negative correlations. A reason for this could be the restriction of GitHub utilising only one main feedback system that can be collated through the API whereas law eight refers to ‘multi-level, multi-loop, multi-agent feedback systems’ which shows a disparity between open source development and traditional software teams which can leverage much more resources in terms of feedback. In conclusion based on the results attained in this study law seven as devised by Lehman does not appear to hold.

|  |  |
| --- | --- |
| Lag Interval | Mean Cross Correlation for #issue comments and #issues |
| 0 | -0.005021707208996858 |
| -1 | -0.0010096229736336762 |
| -2 | -0.0058765360127318875 |
| -3 | 0.0013696850546545927 |
| -4 | -0.0016452829192041823 |
| -5 | -0.0012483542309469358 |
| -6 | 1.813146532671201E-4 |
| -7 | 0.00707484489909087 |
| -8 | -0.0016174230773226398 |
| -9 | 0.0022468901712385096 |

Figure 18 – mean correlation for issues comments and issues at different lag intervals

Figure 19 – distribution of cross correlation values at different lag intervals

**VI. THREATS TO VALIDITY**

**A. Construct Validity**

Initial hypothesis generation will be examined, 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 study. In addition to this law 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 more abstract term that could account for testing code coverage, architecture, count of bugs among others but due to the restrained of utilising only API produced data, this was a good option that captured the essence of the law which was the main goal when generating hypotheses.

The evaluation process for each hypothesis should also be taken into account, for HP1, 5, 6 & 7 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 taken into account.

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 is trimmed from the evaluation to account for projects migrating to GitHub and the initial dump of data associated with this 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.

**B. External Validity**

Threats to the external validity of the findings also will need to be examined, particularly if the results from this paper can be generalised to open source projects on GitHub in general. Despite the selection of a fairly large set of projects there is no evidence to suggest that the results will remain consistent when applied to a totally different dataset, however due to the paper targeting the most popular projects on GitHub it can be seen as representation of typical open source development for well supported projects not necessarily those that have reduced attention from users.

**C. Conclusion Validity**

In most cases it is difficult to directly support or refute a law rather the main contribution of this paper is a discussion based on the results whether the law appears to hold or is refuted. Therefore a key facet has been a focus on discussion and making inferences based on the data, in some cases (HP2 and HP3) the results provide an overwhelming indication of either supporting or refuting a law. However the other hypotheses remain much more up for debate, the goal of the paper is to determine the validity of Lehman’s laws and each hypotheses has been evaluated with either supporting the law, or not supporting the law, therefore in this context the conclusions drawn are reasonable.

**VII. CONCLUSION**

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 the findings, based on results from a dataset of 100 open source projects, only one of the hypotheses provide enough evidence to support the laws while the other six challenge the validity of the laws they represent. 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. 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.

**VIII. REFERENCES**

* 1. Lehman, Meir M. (1980). "Programs, Life Cycles, and Laws of Software Evolution". Proc. IEEE 68 (9): 1060–1076.
  2. Eirini Kalliamvakou et al, The Promises and Perils of Mining GitHub, MSR 2014 Proceedings of the 11th Working Conference on Mining Software Repositories, Pages 92-101
  3. "GitHub Press Info". https://github.com/about/press. GitHub. Retrieved 7 April 2016.
  4. Georgios Gousios et al, An Exploratory Study of the Pull-based Software Development Model ,MSR 2014 Proceedings of the 11th Working Conference on Mining Software Repositories, Pages 384-387
  5. Lehman, M. M. (1980). "On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle". Journal of Systems and Software 1: 213–221.
  6. Liguo Yu and Alok Mishra (2013) An Empirical Study of Lehman’s Law on Software Quality Evolution in International Journal of Software and Informatics, 11/2013
  7. Jyoti Sheoran et al, Understanding "watchers" on GitHub, MSR 2014 Proceedings of the 11th Working Conference on Mining Software Repositories, Pages 336-339
  8. Jesus M. Gonzalez-Barahona et al, Studying the laws of software evolution in a long-lived FLOSS Project, JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS J. Softw. Evol. and Proc. 2014; 26:589–612
  9. Xu Ben, Shen Beijun,Yang Weicheng, Mining Developer Contribution in Open Source Software Using Visualization Techniques, Intelligent System Design and Engineering Applications (ISDEA), 2013 Third International Conference, 16-18 Jan. 2013, 934 – 937
  10. Ioannis Skoulis et al, Open-Source Databases: Within, Outside, or Beyond Lehman's Laws of Software Evolution?, 26th International Conference on Advanced Information Systems Engineering (CAiSE 2014), At Thessaloniki, Hellas
  11. M.M. Mahbubul Syeed, Imed Hammouda, Tarja Syst¨a, Evolution of Open Source Software Projects: A Systematic Literature Review, Journal of Software, Vol 8, No 11 (2013), 2815-2829, Nov 2013
  12. Hudson Borges, Marco Tulio Valente, Andre Hora, Jailton Coelho, On the Popularity of GitHub Applications: A Preliminary Note
  13. Tegawende F. Bissyand ´ e, Got Issues? Who Cares About It? An Investigation of Issue Trackers of 105 Projects, DOI: 10.1109/ISSRE.2013.6698918 Conference: Software Reliability Engineering
  14. Language Trends on GitHub, https://github.com/blog/2047-language-trends-on-github, August 19, 2015
  15. WG\_LCP - Working Group for Life Cycle Processes, Oversight Committee: C/S2ESC - Software & Systems Engineering Stand, 14764-2006 - ISO/IEC International Standard for Software Engineering - Software Life Cycle Processes – Maintenance
  16. <https://developer.github.com/v3/activity/starring/>
  17. Yan Dong, Shahab Mohsen, Does Firefox obey Lehman’s Laws of software Evolution?
  18. Guowu Xie et al, Towards a Better Understanding of Software Evolution: An Empirical Study on Open Source Software, Software Maintenance, 2009. ICSM 2009. IEEE International Conference on 20-26 Sept. 2009
  19. Iulian Neamtiu et al, Towards a better understanding of software evolution: an empirical study on open-source software, JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS J. Softw.: Evol. and Proc. 2013; 25:193–218
  20. MM Lehman, JF Rami, Software evolution in the age of component-based software engineering, IEE Proceedings-, 2000 - ieeexplore.ieee.org
  21. A Israeli, DG Feitelson, The Linux kernel as a case study in software evolution, Journal of Systems and Software, 2010
  22. E. Stroulia, R. Kapoor, Metrics of Refactoring-based Development: An Experience Report, OOIS 2001, 2001 – Springer
  23. E Karch, Lehman's Laws of Software Evolution and the Staged-Model, 2011 - blogs.msdn.com
  24. Graylin Jay et al, Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship, J. Software Engineering & Applications, 2009, 2: 137-143

**IX. APPENDIX**

This appendix contains the research paper which has been submitted to the ISCME (http://icsme2016.github.io/) conference on software maintenance and evolution. The research paper has been distilled from this dissertation with the assistance of Dr Desmond Greer, the notification date will exceed the dissertation submission deadline therefore the outcome is unknown, however we remain confident the research in some form will be accepted to a conference (if not ISCME) and feel the novel approach contributes to the field.

Do Lehman’s Laws of Software Evolution apply to Open Source Software: A study using GitHub API?

Jordan McDonald and Des Greer

School of Electronics, Electrical Engineering and Computer Science

Queens University Belfast

Belfast, 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 have changed very significantly 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, while 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 each of Lehman’s laws is evaluated using the data available. The study has discovered that the evidence does not support many of the laws, but with the proviso that the study is limited by 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.***

Introduction

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. 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**Error! Reference source not found.**. The goal of this paper is to examine these laws in the context of modern open source projects hosted on GitHub, with a dataset mined via the GitHub API. GitHub is a hosting service designed for collaboration on a centralised 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 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. Users also have profiles that can be populated with identifying information and contain their recent activity within the site[2].

At the time of writing, GitHub reports having 14 million users and 35 million repositories [3] confirming it as the largest host of source code in the world [4]. This coupled with the 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 Lehman’s laws hold. The next section will provide some background to the study, Section II will describe the background to the work, Section III will describe the methodological aspects of the study, Section IV will discuss the results of the work while section V will identify threats to validity in the study. Finally, in Section VI we provide attempt to draw conclusions from the work and identify further work.

BACKGROUND AND RELATED WORK

*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 [1]. 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”*[5]. Table 1 provides a summary of each law.

Lehman’s laws of software evolution

|  |  |
| --- | --- |
| **Law** | **Law Description** |
| 1 | (1974) "**Continuing Change**" - an E-type system must be continually adapted or it becomes progressively less satisfactory |
| 2 | (1974) "**Increasing Complexity**" - as an E-type system evolves, its complexity increases unless work is done to maintain or reduce it |
| 3 | (1974) "**Self-Regulation**" - E-type system evolution processes are self-regulating with the distribution of product and process measures close to normal |
| 4 | (1978) "**Conservation of Organisational Stability (invariant work rate)**" - the average effective global activity rate in an evolving E-type system is invariant over the product's lifetime |
| 5 | (1978) "**Conservation of Familiarity**" - as an E-type system evolves, all associated with it, developers, sales personnel and users, for example, must maintain mastery of its content and behaviour to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves. |
| 6 | (1991) "**Continuing Growth**" - the functional content of an E-type system must be continually increased to maintain user satisfaction over its lifetime |
| 7 | (1996) "**Declining Quality**" - the quality of an E-type system will appear to be declining unless it is rigorously maintained and adapted to operational environment changes |
| 8 | (1996) "**Feedback System**" (first stated 1974, formalised as law 1996) - E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base |

*Related Work*

Attempts at general data mining from GitHub has been prominent in recent years, Kalliamvakou et al.[2] 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 exists a systematic literature review into the evolution of open source projects [6], where the authors examine the data sets utilised, sources of the data and research trends in recent years. The authors found that Lehman’s laws do not hold in certain cases, with individual laws in the research yielding contradicting results in regards to open source projects.

Additional papers have provided much more focused studies, Yu and Mishra [7] investigated the quality of evolving software by mining bug reports and provide evidence to confirm law 7. Sheoran et al [8] investigate “watchers” on GitHub. The paper hones in on 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 Xu Ben et al [9] 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. [4] 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. [10] provide insight into what constitutes project popularity on GitHub making use of the starring mechanism. [11] 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 [12]. 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 [13] 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. [14] 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. This study found that several of Lehman’s laws were supported.

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 [15] . This study uses a single project case study and concludes that the laws apply in this context.

Evaluating Lehman’s laws using data from the GitHub API has not, to the authors’ knowledge, prior to this been fully investigated. Prior studies that are similar to the approach in this paper have flaws 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.

Methodology

*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 (null) hypothesis is also accompanied by justification and reasoning which explain the choice of metric used to represent each law.

**RQ** - Is it possible using data extracted from the GitHub API to determine if OS software evolution over time reflects Lehman’s laws?  
**H1** – If the number of commits decreases the number of star gazers will also reduce (laws 1 & 6).  
**H10** – 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.

**H2** – Total lines of code increases as a software system evolves (law 2).  
**H20** – 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 [16] 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.

**H3** – Issues, additions and deletions over time for are normally distributed (law 3).  
**H30** – 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.

**H4** – As software evolves the quantity of changes to lines of code should remain invariant over time (law 4).  
**H40** – As software evolves the quantity of changes to lines of code will not remain invariant over time  
**Justification** – In this case the ‘global activity rate’ is represented by LOC which reflects the entirety of the project and its contents, therefore it sufficiently meets the ‘global’ requirement as all modifications to the source code will impact the LOC. In addition to this if the phrase ‘work rate’ is considered this in a software context relates closely to changes to the code base and as a consequence of this the LOC which reinforces the selection of this measure.

**H5** – As Lines of code increases the number of issues will also increase (law 5).  
**H50** – There is no relationship between Lines of code and the number of issues.  
**Justification** – To represent an invariant growth of a project lines of code is an ideal choice but an assumption is inherently made that negative growth (reduction in LOC) will have a positive impact on the project. It is possible that a significant decrease in LOC as a result of refactoring may introduce more issues and reduce mastery, but to capture the essence of the law an increase in growth is the focal point of this hypothesis. Issues has been chosen to account for the ‘mastery’ requirement, if a developer has reduced familiarity as a result of significant growth it may lead to increased issues/bugs in the future due to misinterpreting new behaviour or functionality. Issue count has been chosen over stargazer count to represent ‘satisfactory evolution’ as significant growth that exceeds the invariant could be interpreted by passive stargazers as a thriving project which is well supported and thus will continue to stargaze and may even lead to an increase in stargazers who note this activity.

**H6** - Project issues increase as code churn decreases (law 7).  
**H60** – 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.

**H7** – As the number of issue comments increases the number of issues should decrease (law 8).  
**H70** – There is no relationship between the number of issue comments and the number of issues.  
**Justification** – Issues and Issue comments are the main form of feedback in GitHub. Additionally, issues can be used to measure a degree of improvement, an assumption being made that a reduction in issues infers improved code.

*Project Selection*

Add the project to the selection for analysis

Identify the top ten languages

Search each language for the most popular project (sorted by #stars)

Does the project have a fifty percent plus affinity to the target language & metrics fully populated?

Select another programming language

If total projects for the current language equals twenty?

Has the current project been on GitHub for five years or more?

GitHub Project Selection Process

Fig. 1 demonstrates the project selection process. The ten programming languages have been chosen based on a ranking system [17]. 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.

*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 [18] which was integrated into the workbench.

URL(s)

JSON data

DB Query

GitHub API

Webpage

Java Servlet

JSON extractor module (JS)

Raw   
data

MongoDB

R Environment

GitHub Extraction Workbench Architecture

*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 **Error! Reference source not found.**.
* 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.

*Statisitcal Analysis*

*Growth Rate:*

This concept has significant relevance. We use it in tandem with an LOC metric to test H2. In addition growth rate is applied in testing H5, but in this case it will be mapped to a time series with a cross correlation with issues. We can express growth rate and average growth rates as percentages.

Growth Rate -

Average Growth Rate -

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 H3, 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 H1, H5, H6 and H7 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.

*Variance & Standard Deviation*

Law four concerns itself with an invariant work rate, this can be interpreted to applying a variance on the growth rate of the projects LOC. The growth rate will become a series of growth rate values between each weekly LOC and variance will be calculated for this series.

Results and DISCUSSION

*H1: 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 II 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.

percentage positive cross correlation at different lag intervals between #commits and #stargazers

|  |  |
| --- | --- |
| **Lag Amount** | **Percentage of 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% |

The next step will be to consider the significance of the percentage value towards accepting or discarding the null hypothesis. The value itself 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. Fig. 3 visualises the distribution of correlation values at different lag intervals, highlighting the almost random nature obtained, thus supporting the null hypothesis. Table II shows the mean correlation values for each lag interval and while all remain positive the significance of this evidently minor. Therefore, using the evidence available from our chosen metrics, we cannot reject the null hypothesis

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.

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 |

Distribution of correlation for the two extreme lag intervals

*H2 – 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. Fig. 4 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 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 H20, providing support for law two.

Number of projects whose LOC increased or decreased over time

*H3 - 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 IV shows the overall results showing clearly that we cannot reject the H30 hypothesis. Rather, evidence points to the non-normality of the data and counters law 3 despite threshold relaxations from the traditional 0.05. We also conducted an Anderson-Darling statistical test for normality and found a similar result. The 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.

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% |

*H4 - As software evolves the quantity of changes to lines of code should remain invariant over time*

Distribution of LOC growth rate variance for each of the 100 projects

LOC was again chosen as the best metric from which to calculate growth rate, thus creating a vector of percentage growth rate values. The variance of each vector was then derived as shown in Fig. 5. From these graphs it is clear that the growth rate variance for each project can fluctuate between different extremes; the highest values are representative of projects whose growth is unpredictable, possibly due to sudden significant shifts in growth or may have long periods with no change to growth rate that precede a spike in contributions. It is possible to observe significant outliers that are prominent in the set of variance values, therefore to aid in interpretation the median of these values was calculated as 30.290. This needs further analysis and investigation as it may reflect the practice of sudden dumping of code into a repository in some projects,

Because of so many outliers, it is difficult to test the null hypothesis. Hence, to represent a reasonable invariant growth rate the standard deviation for each projects growth rate vector was calculated. Since the amount of lines of code that change per weekly interval may vary based on a number of factors, introducing the standard deviation as to act as a threshold to determine a reasonable distance from the mean may prove useful. This measure would enable determining the percentage of growth rate values that are within one standard deviation distance from the mean growth rate value for each project – see Fig. 6. The vast majority of each projects vectors are showing significant affinity to the one standard deviation invariant work rate threshold which suggests that the over the course of the projects life cycle the lines of code changes remain within a reasonable level of invariance.

Overall, the test for H40 is rather inconclusive and offers little support for Lehman’s fourth law.

% of each projects growth rate values within one standard deviation

*H5 - As Lines of code increases the number of issues will also increase*

Law five "Conservation of Familiarity" suggests that excessive growth of software as time progresses will reduce the mastery of the user base and lead to reduced satisfaction. LOC was again used to represent growth and issues has been utilised as an indicator of user mastery of the software which can then be applied in a cross correlation. Table V shows the results giving a percentage of cross correlation values for each project at different lag points that show a positive correlation.

% of cross correlation values for each project showing a positive correlation

|  |  |
| --- | --- |
| **Lag Amount** | **Percentage of positive correlations \*\*** |
| 0 | 50% |
| -1 | 50% |
| -2 | 49% |
| -3 | 46% |
| -4 | 51% |
| -5 | 44% |
| -6 | 41% |
| -7 | 44% |
| -8 | 50% |
| -9 | 45% |

Initially a discussion will be made on the impact of applying a lag to the LOC growth rate has on its correlation with the amount of issues generated by users for the projects. Based in particular on the -8 lag result in comparisons to those which represents the impact of a change in LOC in weeks closer to the ‘present’ point for issues there appears to be no point that an increase/decrease in growth rate has an impact on the amount of issues. Reasons for this could include the sporadic nature of growth in open source projects which often do not confirm to a schedule, new code is often integrated on an ad-hoc basis and if a change introduces any problems (an issue) it may only become evident in a very specific use case at an arbitrary point in time before being reported. In addition to this, Lehman focuses on ‘familiarity’ and ‘mastery’ which are not terms immediately applicable to the dynamic pool of contributors for an OSS project *outside* the core development team who may develop code base *without* in-depth knowledge of the whole software product.

mean correlation value at each lag interval for LOC Growth Rate and #Issues

|  |  |
| --- | --- |
| **Lag Interval** | **Mean Cross Correlation for LOC growth rate and #Issues** |
| 0 | -0.009 |
| -1 | 0.005 |
| -2 | 0.004 |
| -3 | -0.013 |
| -4 | -0.014 |
| -5 | 0.008 |
| -6 | 0.008 |
| -7 | 0.005 |
| -8 | 0.003 |
| -9 | 0.007 |

Distribution cross correlations of Commits and Stars at different lags

The results shown in Table V do not show a strong relationship between the growth rate and the number of issues. Only one of the outcomes produce a majority positive correlation. Thus the null hypothesis cannot be rejected and law five is not supported in this study. To reinforce this point a series of graphs has been presented in Fig. 7 which highlight the lack of relationship and random nature of the correlation values attained. In addition to this Table VI presents the mean correlation for each lag interval, where the means show only a minor preference to positive or negative values.

*H6 - 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 VII.

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% |

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.

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 V suggests that changing 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.

Distribution of correlations at different lag points

To conclude reference should be made back to the hypothesis to determine an outcome, the results in Table VII provide evidence which disputes law 7 in most cases. Fig. 8 shows a distribution of values that appear random indicating little support for H6. Further, this claim which albeit contrast with the significant negative presence as seen in Table VII shows only a minor affinity towards a positive correlation in the case of a mean.

*H7 – As the number of issue comments increases the number of issues should decrease*

Law seven focuses on a ‘feedback system’ and its effect on improving the code base, the major feedback facility on GitHub can be seen in issue tracking which allows users to report bugs and discuss new features and thrives on interaction in the form of comments to enable discussion. Based on this issue comments have been selected to represent the feedback system and issues itself will represent code improvement with a decrease in issues seen as a key factor as the amount of comments increase. Once again a cross correlation has been chosen as the measure to evaluate this hypothesis, with the percentage of negative correlations at each tested lag interval used to determine if the amount of comments indeed reduce the amount of issues.

Percentage of correlations that are negative for different lag points

|  |  |
| --- | --- |
| **Lag Amount** | **Percentage of negative correlations** |
| 0 | 53% |
| -1 | 61% |
| -2 | 49% |
| -3 | 53% |
| -4 | 59% |
| -5 | 60% |
| -6 | 61% |
| -7 | 65% |
| -8 | 64% |
| -9 | 69% |

If the results in Table VIII are observed there lies relationship between the size of the lag interval and the percentage of projects that output a negative correlation for these two metrics. Initially consider the zero lag which measures the effect of comments on the count of issues that occur in the same week. This shows a lower proportion of negative correlation and this could be attributed to the fact that discussion is ongoing about a particular issue and therefore there is less of an opportunity to tackle the problem. This is a side effect of open source projects coordinating distributed teams which a certain amount of delay is to be expected due to the medium of communicating via a comments system, reducing the scope for rapid solutions to an issue. This is generally reflected in the majority of the smaller lag intervals. In contrast the more the lag applied, the more likely that the number of comments will drive the volume of issues.

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.

Mean correlation for issues comments and issues at different lag intervals

|  |  |
| --- | --- |
| **Lag Interval** | **Mean Cross Correlation for #issue comments and #issues** |
| 0 | -0.005 |
| -1 | -0.001 |
| -2 | -0.006 |
| -3 | 0.001 |
| -4 | -0.002 |
| -5 | -0.001 |
| -6 | 0 (1.8E-4) |
| -7 | 0.007 |
| -8 | -0.002 |
| -9 | 0.002 |

Distribution of cross correlation values at different lag intervals

In terms of evaluating H7 the initial evidence across the 100 projects suggests rejection of the null hypothesis. However this support for H7 is tempered when considering the variation when different lags are considered and the fact that the dispersion of the correlations for each of the one hundred projects appears random (Table IX and Fig. 9). Fig. 9 shows the mean correlation value at each lag for each of the one hundred projects and there seems to be no significant affinity to either positive or negative correlation. A reason for this could be the restriction of GitHub utilising only one main feedback system that can be collated through the API whereas law eight refers to ‘multi-level, multi-loop, multi-agent feedback systems’ which shows a disparity between open source development and traditional software teams which can leverage richer resources in terms of feedback

THREATS TO VALIDITY

*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.

The evaluation process for each hypothesis should also be taken into account, for H1, H5, H6 & H7 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.

*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.

*Conclusion validity*

Using the results, in most cases it is difficult to directly 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. H2 and H3) the results provide clear support or refutation However the other hypotheses remain much more debateable.

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 the findings, based on results from a dataset of 100 open source projects, only one of the hypotheses provide enough evidence to support the laws while the other six challenge the validity of the laws they represent. 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. 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

M.M. Lehman, "On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle," Journal of Systems and Software vol. 1, pp. 213–221, 1980.

E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D.M. German & D. Damian, “The promises and perils of mining github”. *Proc. 11th working conference on mining software repositories,* pp. 92-101, 2014.

"GitHub Press Info". https://github.com/about/press. GitHub. Retrieved 7 April 2016.

Georgios Gousios et al, An Exploratory Study of the Pull-based Software Development Model ,MSR 2014 Proceedings of the 11th Working Conference on Mining Software Repositories, Pages 384-387.

M M. Lehman, "Laws of software evolution revisited." Software process technology, pp. 108-124. Springer Berlin Heidelberg, 1996.

M. Syeed, I. Hammouda, and T. Syatä, “Evolution of Open Source Software Projects: A Systematic Literature Review”, Journal of Software, Vol 8, No. 11, pp.2815-2829, Nov 2013.

L. Yu and A. Mishra, “An Empirical Study of Lehman’s Law on Software Quality Evolution” International Journal of Software and Informatics, Vol. 7, No. 3, pp. 469-481, 2013.

J. Sheoran, K. Blincoe, E. Kalliamvakou, D. Damian, and J. Ell, Understanding "watchers" on GitHub, MSR 2014 Proc. 11th Working Conference on Mining Software Repositories, pp. 336-339, 2014.

X. Ben, S Beijun, Y. Weicheng, “Mining Developer Contribution in Open Source Software Using Visualization Techniques”, Intelligent System Design and Engineering Applications (ISDEA), Proc. 3rd International Conference, pp. 934 – 937, 2013.

H. Borges, M.T. Valente, A. Hora, J. Coelho, “On the Popularity of GitHub Applications: A Preliminary Note” arXiv preprint arXiv:1507.00604 (2015).

Tegawende F. Bissyandé, “Got Issues? Who Cares About It? An Investigation of Issue Trackers of 105 Projects”, Proc. IEEE 24th International Symposium on Reliability Engineering, pp. 188-197, 2013.

J.M. Gonzalez‐Barahona, R. G. Robles, H. I. Herraiz, and F. Ortega. "Studying the laws of software evolution in a long‐lived FLOSS project." Journal of Software: Evolution and Process, Vol. 26, No. 7, pp.589-612. 2014.

I. Neamtiu, G. Xie and J. Chen, “Towards a better understanding of software evolution: an empirical study on open-source software”, Journal of Software: Evolution And Process, Vol. 25, pp.193–218, 2013

I. Skoulis, P. Vassiliadis, A. Zarras. “Open-Source Databases: Within, Outside, or Beyond Lehman’s Laws of Software Evolution?” Proc. 26th International Confernece on Advanced Information Systems Engineering, pp. 379-393, 2014.

E. Stroulia, R. Kapoor, Metrics of Refactoring-based Development: An Experience Report, OOIS 2001, pp. 113-122, 2001.

Graylin Jay et al, Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship, J. Software Engineering & Applications, 2009, 2: 137-143.

“Language Trends on GitHub”, https://github.com/blog/2047-language-trends-on-github, accessed 7 August, 2015.

The R Project for Statistical Computing, <https://www.r-project.org/>, accessed 7 April, 2016.

A Israeli, DG Feitelson, The Linux kernel as a case study in software evolution, Journal of Systems and Software, 2010.

GitHub Developer Reference, “Starrting” <https://developer.github.com/v3/activity/starring/> accessed 7 April, 2016